O'REILLY®

Aprendendo Node

USANDO JAVASCRIPT NO SERVIDOR

novatec

Shelley Powers

Sumário

Prefácio

O Node.js já existe há algum tempo e alcançou um sucesso considerável, tendo sido adotado por nomes expressivos do mercado de tecnologia, como LinkedIn, Yahoo! e Netflix. Contudo, ainda é jovem o bastante para deixar de cabelo em pé o gerente operacional típico, cujo habitat são as grandes empresas. O Node tem sido, ao longo dos anos, uma força motriz imperiosa por trás de um "ecossistema" de aplicações sofisticadas em JavaScript, e se estabeleceu como o único porto seguro no qual podemos usar esse JavaScript renovado, sempre em evolução. E, em perfeita simbiose, esse JavaScript de última geração devolve o favor ao ser o fator decisivo para a nova versão totalmente repaginada do Node.js, com nova organização interna e paradigma de lançamentos.

O Node.js também redefiniu a função no Cosmos para a qual o JavaScript existe. Nos dias de hoje, é muito provável que as empresas exijam de seus desenvolvedores em JavaScript uma desenvoltura tanto no navegador quanto no servidor. Se o desenvolvedor estiver familiarizado apenas com o ambiente do navegador, terá problemas. A novidade de usar uma só linguagem no navegador e no servidor está chamando muita atenção de desenvolvedores vindos de Ruby, C++, Java e PHP – especialmente aqueles que já conhecem JavaScript.

Para mim, o Node.js é diversão pura. Comparado a outros ambientes, o esforço para começar um projeto é mínimo, e continua mínimo durante todo o período de desenvolvimento e implementação em produção. Também é muito simples testar coisas novas. O "esqueleto"[1] inicial necessário para criarmos um projeto em Node é substancialmente menos complexo e exigente do que o esperado para qualquer outra linguagem. Apenas o PHP oferece um ambiente tão despojado quanto o do Node, e mesmo ele requer uma integração bastante rigorosa com o Apache para criar aplicações acessíveis ao público geral.

Mesmo sendo simples, o Node.js tem lá suas partes que podem ser difíceis de descobrir. Por exemplo, para o pleno aprendizado do Node.js é preciso que o desenvolvedor já domine completamente tanto o ambiente em que a aplicação[2] entrará em produção quanto as APIs a serem utilizadas. Depois de dominar esses fatores externos, o programador tem ainda que descobrir onde estão e como funcionam os meandros do próprio Node.

A quem este livro se destina

Vejo dois públicos distintos para este livro.

O primeiro é o desenvolvedor de frontend que vem criando, há tempos, aplicações que rodam direto no navegador do cliente, usando bibliotecas e frameworks como o AngularJS e o jQuery, e quer agora empregar no backend (ou seja, no servidor) esse conhecimento já sedimentado sobre JavaScript.

O segundo público é o desenvolvedor de backend que já cria, usando outras linguagens, aplicações que rodam no servidor e quer experimentar algo novo. Há ainda os que não só querem, mas precisam (por quaisquer motivos técnicos, profissionais ou outros) fazer a transição para uma nova tecnologia. Esse desenvolvedor já é experiente em linguagens típicas de backend, como C++, Ruby ou PHP, mas agora quer empregar no servidor, que conhece intimamente, o JavaScript que foi aprendendo com o tempo enquanto interagia com o pessoal do frontend.

Esses dois públicos aparentemente separados compartilham do mesmo conhecimento básico: a programação em JavaScript – ou, se preferir ser mais específico, ECMAScript, que é o nome oficial da linguagem. Este livro não é, portanto, para iniciantes, pois você já precisa ter alguma experiência no desenvolvimento em JavaScript. Outro ponto em comum entre os dois públicos é que ambos precisam aprender os mesmos princípios básicos do Node, incluindo a API central do Node.

Entretanto, cada público tem perspectivas, bagagem e talentos diferentes, e isso influi na curva de aprendizado. Para que o livro seja o mais útil

possível, tentarei incorporar ambas as perspectivas no material. Por exemplo, um desenvolvedor vindo do C++ ou Java pode ter especial interesse em desenvolver extensões e plugins para o Node em C++, o que nem passa pela cabeça de um desenvolvedor de frontend. Ao mesmo tempo, alguns conceitos como o *big-endian* podem ser bastante familiares para os desenvolvedores de servidor, mas completamente ignorados pelo pessoal do frontend. Não posso me colocar no lugar de cada programador o tempo todo nem tratar desses assuntos com muita profundidade, pois o espaço do livro não permite, mas tentarei garantir que a maioria dos leitores não fiquem frustrados ou aborrecidos.

O que *não* farei é forçá-lo a aturar "decorebas". Por exemplo, veremos as APIs dos módulos nativos do Node, mas não vou ficar listando cada objeto e função. Tudo isso está muito bem documentado no site do Node (*https://nodejs.org/en/*). O que faremos juntos é passear pelos aspectos importantes de cada módulo nativo ou de alguma funcionalidade específica do Node. Acredito que o importante é dotá-lo com o essencial a respeito do Node para que possa trabalhar em conjunto com outros desenvolvedores. A documentação está aí para ser consultada, não decorada. Claro, a prática leva à perfeição, e este livro é uma ferramenta de aprendizado. Ao terminá-lo, é preciso continuar a jornada com explorações cada vez mais aprofundadas a respeito de funcionalidades específicas, como, por exemplo, a programação em full-stack usando o MEAN (a famosa pilha de tecnologias formada por MongoDB, ExpressJS, AngularJS e Node.js). Este livro é a semente inicial que, plantada, nos permitirá escolher no futuro qual ramificação no Node queremos seguir.

Uma palavra sobre a documentação do Node

No momento em que este livro estava sendo escrito, um grupo grande de desenvolvedores de Node, no qual eu me incluía, estava em calorosa discussão sobre diversos problemas encontrados no site oficial do Node.js. Entre eles estava definir claramente qual era a versão "atual" do Node.js, que é uma das primeiras coisas que devem estar documentadas quando alguém acessa a mãe de todas as documentações – ou seja, a oficial.

Na última vez que participei dessa discussão, o plano era listar todas as versões do Node.js na página */docs*, tanto as versões antigas com suporte estendido (Long-Term Support – LTS) quanto a versão estável mais atual (Current Stable), e providenciar um indicador para a documentação apropriada de cada versão no topo de toda e qualquer página da

documentação.

Em algum momento no futuro, o pessoal da documentação vai gerar *arquivos diff* com as diferenças entre versões das APIs para cada página, mas esse será um projeto desafiador.

Quando este livro foi publicado, a comunidade Node lançou o Node.js 6.0.0 como versão Current e abandonou a designação Stable, que anteriormente era associada a ela.

Por conta dessa salada de versões, sempre que acessar a documentação de APIs no site oficial do Node.js, certifique-se de usar a documentação apropriada para a versão de Node.js que estiver usando. Aproveite para verificar a documentação das versões mais novas, para se inteirar das novidades que usará em seu próximo projeto.

O Node.js é apenas Node. Ponto!

O nome formal do software é Node.js, mas ninguém o chama por esse nome! Todo mundo usa apenas "Node". Fim de papo. E é o que vamos usar, na maior parte do tempo, neste livro.

Estrutura do livro

Apesar de não ser um livro para iniciantes, em relação ao Node.js o *Aprendendo Node* é um livro do tipo bê-á-bá. Ele se concentra nos fundamentos primordiais do Node e de seus módulos nativos (ou seja, que vêm junto com ele, não são de terceiros). Falaremos brevemente sobre módulos de terceiros e discutiremos extensamente sobre o npm, é claro, mas o objetivo primário deste livro é fazer com que o leitor entenda os conceitos básicos do funcionamento do Node. A partir dessa plataforma sólida podemos construir estruturas mais complexas e desafiadoras.

O Capítulo 1 introduz o Node, incluindo uma breve seção sobre como instalá-lo. Teremos a chance de levar o Node para dar uma volta, primeiro criando um servidor web com o famoso código indicado na documentação oficial do Node. Depois, criaremos um servidor um pouco mais avançado usando um código deste livro. Veremos também como criar plugins (na nomenclatura do Node, add-ons) usando C e C++. E o que seria de uma introdução ao Node sem uma introdução à história desse ambiente, cuja primeira versão não foi 1.0, mas sim 4.0?

O Capítulo 2 mostra a funcionalidade essencial do Node, incluindo manipulação de eventos, objetos globais e seu uso na criação de aplicações e a natureza assíncrona do Node. Também veremos o objeto

buffer, a estrutura de dados transmitida pela maioria dos serviços de rede do Node.

O Capítulo 3 mergulha na natureza modular do Node, bem como no npm, um sistema de administração de módulos. Apesar de não ser nativo, mas sim de terceiros, o npm é usado por dez entre dez desenvolvedores para instalação e gerenciamento de módulos em um projeto. Nesse capítulo, veremos como a aplicação em desenvolvimento encontra os módulos do Node que está usando e como criar seu próprio módulo. Por fim, discutiremos alguns dos aspectos avançados da funcionalidade do Node, usando para isso o sandbox (ambiente seguro para testes). Para nos divertirmos um pouco, também veremos três módulos de terceiros muito populares para o Node: Async, Commander e Underscore.

O console interativo que vem com o Node, que atende pelo nome de REPL, é uma ferramenta de aprendizado valiosíssima e, por isso mesmo, terá um capítulo exclusivo – o Capítulo 4. Veremos como usar a ferramenta, em detalhes, bem como maneiras de criar seu próprio REPL personalizado.

Começaremos a construir aplicações web em Node no Capítulo 5 e aproveitaremos para conhecer de perto alguns módulos do Node que facilitam o desenvolvimento web. Teremos a chance de ver com quantos paus se faz um servidor web estático e completamente funcional, e também aprenderemos como executar uma aplicação Node na mesma máquina em que já exista um Apache, pelo uso de um proxy do próprio Apache.

O Node funciona em um sem-número de ambientes, incluindo o Windows e sistemas baseados no Unix, como o Mac OS X e o Linux. Isso só é possível porque o Node tem uma funcionalidade corretiva que padroniza quaisquer diferenças entre sistemas. Essa funcionalidade será explorada no Capítulo 6. Também veremos a natureza fundamental dos fluxos de dados e encadeamento de entrada e saída (no jargão do Node, respectivamente, *streams* e *pipes*, que são os elementos essenciais de qualquer operação de entrada e saída) e exploraremos o suporte ao sistema de arquivos do Node.

O Capítulo 7 discorre sobre redes. Não é possível abordar esse tema sem tropeçar em outro de igual importância – a segurança. São dois assuntos que devem andar de mãos dadas o tempo todo, como arroz e feijão, goiabada e queijo branco ou chocolate com qualquer coisa que exista no Universo. Veremos o básico sobre TCP e UDP e sua relação com o Node, bem como uma maneira de implementar um servidor HTTPS que rode em paralelo ao servidor HTTP implementado no Capítulo 5. Também discutiremos a mecânica por trás dos certificados digitais e os fundamentos do Secure Sockets Layer (SSL) e seu sucessor, o Transport Layer Security (TLS). Por fim, conheceremos o módulo de criptografia do Node, que empregaremos para cifrar senhas e hashes.

Uma de minhas funcionalidades preferidas do Node é a capacidade de interagir com o sistema operacional por meio de *processos-filho*. Por exemplo, gosto muito de duas pequenas ferramentas escritas em Node. Uma delas é um utilitário para trabalhar com arquivos compactados. A outra interage com o programa ImageMagick, normalmente instalado nos Linux mais populares, e pode com isso capturar telas de websites. Não são aplicações enormes com interfaces sofisticadas na nuvem, mas são muito divertidas e ótimos exemplos para aprender a usar processos-filho. Falaremos sobre eles no Capítulo 8.

Muitos dos exemplos do livro usam o mesmo JavaScript ao qual estamos habituados há anos. Entretanto, um dos motivos que levou à criação do io.js, uma dissidência do Node.js, e à posterior reunificação dos dois em um produto só na versão 4.0, foram diferenças de opinião quanto ao suporte às novas versões do ECMAScript, como a atual ES6 (ou, se preferir, ECMAScript 2015). No Capítulo 9, veremos o que é atualmente suportado no Node, o impacto da nova funcionalidade e quando (e por que) usar a nova funcionalidade em lugar da velha. Também mostraremos as pegadinhas que aparecem quando usamos o JavaScript mais moderno. Contudo, sempre que possível, usaremos a funcionalidade tradicional e, de fato, a única vez que deixaremos a funcionalidade nativa de lado para usar a nova será quando discutirmos o conceito de promessas (promises) no JavaScript, da forma como foram implementadas pelo popular módulo Bluebird.

O Capítulo 10 aborda os frameworks e suas funcionalidades, que normalmente são usados para implementar sistemas em uma estrutura conhecida como *desenvolvimento em full-stack*. Conheceremos o Express, um componente quase onipresente quando se trata de sistemas sérios desenvolvidos em Node. Acima dele, veremos dois bancos de dados, MongoDB e Redis. Exploraremos também dois frameworks de frontend que completam a pilha em soluções full-stack tendo o Node como base: AngularJS e Backbone.js.

Terminada a aplicação em Node, vamos querer enviá-la para o ambiente de produção. O Capítulo 11 descreve as ferramentas e técnicas necessárias para desenvolvimento e implementação em produção de soluções Node. O pacote inclui testes de unidade, carga e desempenho (benchmark), bem como as ferramentas e informações essenciais. Veremos também como deixar um servidor de aplicações Node em funcionamento por toda a eternidade e como rearmá-lo automaticamente caso trave devido a uma falha.

O Capítulo 12 é a sobremesa. Nesse capítulo, levaremos seus conhecimentos de cientista louco do Node para outros planetas, incluindo versões do Node que rodam em microcontroladores, versões que rodam em objetos ligados à Internet das Coisas (IoT – Internet of Things) e uma versão do Node que não roda em V8.

Convenções usadas neste livro

As convenções tipográficas a seguir foram usadas neste livro:

Itálico

Indica novos termos, URLs, endereços de email, nomes ou extensões de arquivo.

`Largura constante`

Usada em listagens de programas, e também no corpo dos parágrafos para se referir a elementos do programa, como variáveis, funções, bancos de dados, tipos de dados, variáveis de ambiente, instruções[3] e palavras-chave.

`Largura constante em negrito`

Mostra comandos ou quaisquer textos que devam ser digitados pelo usuário.

`Largura constante em itálico`

Mostra algum texto que deva ser substituído por valores especificados pelo próprio usuário, ou por valores determinados pelo contexto.

Este elemento significa uma dica ou sugestão.

Este elemento indica uma observação geral.

Este elemento indica uma situação que requer atenção ou cautela.

Usando os códigos de exemplo de acordo com a política da O'Reilly

O material suplementar (exemplos de código, exercícios etc.) está disponível para download em *https://github.com/shelleyp/LearningNode2*.

Este livro existe para que você possa fazer o seu trabalho. Em geral, os códigos oferecidos como exemplo neste livro podem ser usados como parte de quaisquer programas e documentação que você venha a criar. Não é necessário entrar em contato com a O'Reilly, a Novatec ou a autora, a não ser que você esteja copiando uma quantidade significativa de código. Por exemplo, escrever um programa que use dezenas (ou mesmo centenas) de trechos de código deste livro não precisa de permissão. Por outro lado, vender ou distribuir um CD-ROM contendo todos os exemplos deste ou de outros livros da O'Reilly requer permissão expressa. Responder a uma pergunta de alguém citando este livro e incluindo na citação um trecho de código não precisa de permissão, mas incluir uma quantidade significativa dos exemplos aqui contidos na documentação do seu produto certamente requer permissão explícita.

Embora não seja obrigatório, apreciaríamos muito se quaisquer citações ou uso de material contido neste livro tivesse o devido crédito impresso.

Esse crédito inclui o título, autor, editora e ISBN. Por exemplo: "*Learning Node*, por Shelley Powers (O'Reilly). Copyright 2016 Shelley Powers, 978-1-4919-4312-0".

Caso perceba que seu uso dos exemplos de código não se enquadrem como "fair use" (uso justo) ou na permissão descrita aqui, por favor nos contacte: *permissions@oreilly.com*.

Como entrar em contato conosco

Envie seus comentários e suas dúvidas sobre este livro à editora escrevendo para: *novatec@novatec.com.br*.

Temos uma página web para este livro na qual incluímos erratas, exemplos e quaisquer outras informações adicionais.

- Página da edição em português:

  *http://www.novatec.com.br/livros/aprendendo-nod*e

- Página da edição original em inglês:

  http://bit.ly/learning-node-2e

Para obter mais informações sobre os livros da Novatec, acesse nosso site: *https://novatec.com.br*.

Agradecimentos

Gostaria de agradecer ao pessoal que gentilmente me ajudou a montar este livro: à editora Meg Foley, ao revisor técnico Ethan Brown, às revisoras Gillian McGarvey e Rachel Monaghan, à conferente Judy McConville, à ilustradora Rebecca Panzer e a todos que deram uma mãozinha na realização desta obra!

---

1 N. do T.: No original, "scaffolding", cuja tradução literal é "andaime". Em português, neste contexto, "esqueleto" faz mais sentido.

2 N. do T: Em inglês, temos apenas a palavra "application". Em português, havia uma antiga convenção para a tradução dessa palavra, na qual aplicação era o problema a ser resolvido, e aplicativo (ou programa aplicativo, em jargão ainda mais obsoleto) era o software propriamente dito. Essa distinção se perdeu ao longo dos anos, e hoje usa-se aplicação indiscriminadamente. Um fantasma dessa convenção permanece de forma curiosa nos apps para dispositivos móveis,

que consideramos um termo masculino: o app.

3 N. do T.: Em português, *statement*, *declaration* e *assertion* costumam ser traduzidos para a mesma palavra, *declaração*, mas em programação cada palavra tem um significado diferente. Neste livro, para evitar ambiguidades, traduzimos *statement* como *instrução*, *declaration* como *declaração* e *assertion* como *asseveração*.

CAPÍTULO 1

Ambiente do Node

Esqueça tudo o que já ouviu sobre o Node ser uma ferramenta para usar só no servidor. É bem verdade que ele é usado principalmente em aplicações no servidor, mas pode também ser instalado em qualquer máquina e usado para qualquer fim, permitindo rodar aplicações localmente em seu PC ou mesmo em tablets, smartphones ou microcontroladores.

Tenho o Node instalado em meu servidor Linux, mas também o coloquei em todos os meus PCs com Windows 10 e até em meu microcontrolador Raspberry Pi. Possuo um tablet Android a mais e planejo testar o Node nele, algo que já fiz com meu microcontrolador Arduino Uno, e já estou fazendo experiências para incorporar o Node em meu sistema de automação residencial graças ao canal Makers do serviço IFTTT. No PC uso o Node como um ambiente de testes para o JavaScript e como uma interface para o ImageMagick para editar automaticamente muitas fotos de uma vez só. O Node é minha principal ferramenta para qualquer operação em lote que eu precise fazer, tanto no servidor quanto no PC.

Obviamente, uso o Node também para tarefas típicas de servidor quando preciso de uma interface web independente do meu Apache, ou para providenciar um processo exclusivo para o backend de alguma aplicação web.

O que quero dizer com tudo isso é: o ambiente criado pelo Node é rico em funcionalidade e alcance. Para explorar esse ambiente, temos que começar pelo início: a instalação do Node.

IFTTT

IFTTT é um serviço maravilhoso na nuvem que permite conectar eventos e ações vindos de um número enorme de empresas, serviços e produtos usando uma lógica condicional

simples, do tipo if-then. Cada ponto de conexão de uma receita é um canal, incluindo o já citado Maker Channel (*https://ifttt.com/maker*).

Instalando o Node

O melhor lugar para começar é a página de downloads do Node.js (*https://nodejs.org/en/download/*). Aqui podemos baixar os binários (executáveis pré-compilados) para as arquiteturas Windows, OS X, SunOS, Linux e ARM. A página também oferece instaladores para arquiteturas específicas que podem simplificar drasticamente o processo de instalação – especialmente no Windows. Se seu ambiente é específico para desenvolvimento, o melhor é baixar o código-fonte e compilar o Node diretamente. É como prefiro fazer em meu servidor Ubuntu.

Podemos também instalar o Node usando o gerenciador de pacotes ("loja de software") de sua arquitetura. Ir por esse caminho é útil não apenas para instalar o Node, mas também para mantê-lo atualizado (falaremos mais sobre isso ainda neste capítulo, na seção "Node: a versão LTS e o processo de atualização").

Se decidir compilar o Node localmente em sua máquina, será preciso configurar um ambiente adequado para a compilação, além das ferramentas apropriadas. Por exemplo, no Ubuntu (que é um Linux), precisaremos rodar o seguinte comando no terminal para instalar as ferramentas necessárias ao Node:

```
apt-get install make g++ libssl-dev git
```

Há algumas diferenças de comportamento quando instalamos o Node nas várias arquiteturas. Por exemplo, quando o instalamos no Windows, o instalador não providencia somente o Node, mas também cria um terminal de comandos especial para acessá-lo em sua máquina. O Node é um programa de linha de comando e não tem uma interface gráfica como os aplicativos mais comuns para Windows. Caso queira usá-lo para programar um Arduino, é preciso instalar também o Johnny-Five (*http://johnny-five.io*).

Aceite os defaults do mundo Windows

Sempre aceite o local-padrão de instalação e as opções default quando instalar o Node no

Windows. O instalador adiciona o Node à variável PATH, o que significa que podemos digitar **node** no terminal sem precisar informar o caminho completo.

Caso esteja instalando o Node no Raspberry Pi, baixe a versão apropriada da arquitetura ARM, por exemplo, ARMv6 para o Raspberry Pi original e ARMv7 para o Raspberry Pi 2, mais recente. Uma vez baixado, extraia o binário a partir do arquivo *.tar.gz* e copie a aplicação para */usr/local*:

```
wget https://nodejs.org/dist/v4.0.0/node-v4.0.0-linux-armv7l.tar.gz
tar -xvf node-v4.0.0-linux-armv7l.tar.gz

cd node-v4.0.0-linux-armv7l

sudo cp -R * /usr/local/
```

Sempre podemos, também, configurar um ambiente de desenvolvimento e compilar o Node diretamente.

Node em ambientes inesperados

A respeito do Node no Arduino e Raspberry Pi, discutiremos seu funcionamento em ambientes heterodoxos, como a Internet das Coisas (Internet of Things, IoT), no Capítulo 12.

Diga "olá mundo" com o Node

Você acabou de instalar o Node e naturalmente quer experimentá-lo. Há uma tradição entre os programadores de que seu primeiro programa em uma linguagem seja o famoso "Hello World". O programa, em geral, escreve as palavras "Hello World" no dispositivo-padrão de saída, demonstrando como uma aplicação pode ser criada, executada e a maneira como processa entrada e saída.

O Node não foge à tradição: é esse o programa que o site oficial do Node.js inclui na sinopse da documentação. E também será nossa primeira aplicação neste livro, mas com alguns melhoramentos.

Um Hello World básico

Primeiro, vejamos como é o "Hello World" incluído na documentação do Node. Para recriar a aplicação, digite o código em JavaScript a seguir em um novo documento de texto. Use o editor de texto que preferir. No Windows costumo usar o Notepad++, e no Linux uso o Vim.

```
var http = require('http');

http.createServer(function (request, response) {
  response.writeHead(200, {'Content-Type': 'text/plain'});
  response.end('Hello World\n');
}).listen(8124);

console.log('Server running at http://127.0.0.1:8124/');
```

Grave o arquivo com o nome *hello.js*. Para executá-lo, abra um terminal se estiver no Linux ou no OS X, ou uma janela do Node Command se estiver no Windows. Navegue até o diretório no qual salvou o arquivo e digite o comando a seguir para rodar a aplicação:

```
node hello.js
```

O resultado será mostrado no próprio terminal, graças à chamada à função `console.log()`:

```
Server running at http://127.0.0.1:8124/
```

Agora, abra um navegador e digite **`http://localhost:8124/`** ou **`http://127.0.0.1:8124`** na barra de endereços (se você estiver hospedando o Node em um servidor remoto, digite o domínio apropriado). O que aparecerá é uma página simples e sem graça com o texto "Hello World" logo no início, como mostrado na Figura 1.1.
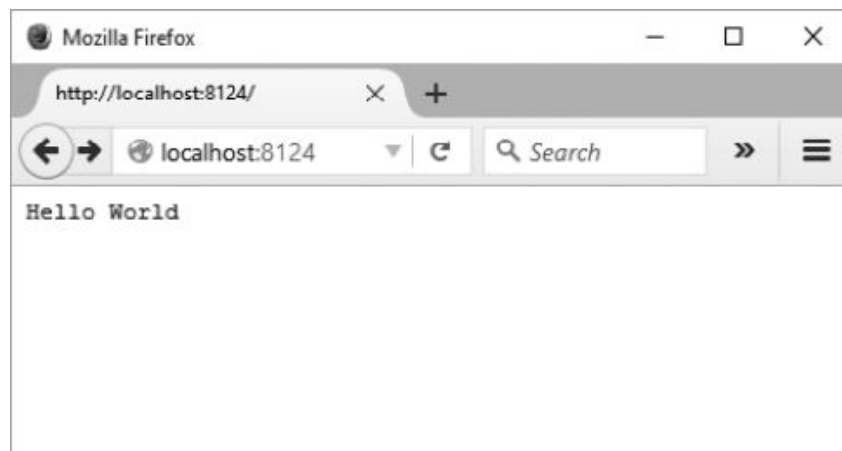
Figura 1.1 – Sua primeira aplicação em Node.

Se estiver rodando a aplicação no Windows, é bem possível que receba um alerta do Windows Firewall, como mostrado na Figura 1.2. Desative a opção para Redes Públicas (Public networks), ative a versão de Redes Privadas (Private networks) e depois clique no botão Permitir Acesso

(Allow access).

Não é necessário repetir esse processo no Windows: o sistema se lembrará da sua escolha.

Para interromper o programa, você pode tanto fechar o terminal, ou a janela de comandos se estiver no Windows (daqui em diante vamos chamar tudo de terminal, ok?), quanto pressionar Ctrl-C. Quando a aplicação foi executada, estava rodando em primeiro plano. Isso significa que não podemos usar o terminal enquanto a aplicação estiver ativa. Também significa que, quando fechamos o terminal, o processo do Node é encerrado.

Figura 1.2 – Permitindo o acesso à aplicação do Node no Windows.

Executando o Node para todo o sempre

Por enquanto, rodar o Node em primeiro plano é o melhor a fazer. Estamos aprendendo a usar a ferramenta, ainda não queremos que a aplicação esteja disponível externamente para qualquer um e queremos poder encerrá-la sempre que necessário. No Capítulo 11, veremos como criar um ambiente de execução mais robusto para o Node.

Voltemos ao código do Hello World. O JavaScript cria um servidor web que mostra uma página, acessada pelo navegador, com as palavras “Hello World”. Esse exercício demonstra uma série de componentes-chave em uma aplicação Node.

Primeiro, inclui o módulo necessário para ativar um servidor HTTP simples, cujo nome, bastante adequado, é HTTP. Alguma funcionalidade que não exista nativamente no Node pode ser incorporada por meio de módulos, exportando funcionalidades específicas que podem ser usadas pela aplicação (ou por outro módulo). O conceito é semelhante ao das bibliotecas em outras linguagens.

```
var http = require('http');
```

Módulos do Node, módulos nativos (core) e o módulo http

O módulo HTTP é um dos módulos nativos (core modules) do Node, que são o objetivo principal deste livro. Veremos mais sobre os módulos do Node, bem como o seu gerenciamento, no Capítulo 3. O módulo HTTP será examinado no Capítulo 5.

O módulo é importado usando a instrução `require` do Node, e o resultado é atribuído a uma variável local. Uma vez importado, sua variável pode ser usada para instanciar o servidor web com a função `http.createServer()`[1]. Nos parâmetros da função, vemos uma das estruturas de código fundamentais do Node: a *função de retorno*, mais conhecida como *callback* (Exemplo 1.1). É essa função anônima que está passando a solicitação web para que o código processe; também é ela que devolve a resposta à solicitação.

Exemplo 1.1 – A função de callback do Hello World

```
http.createServer(function (request, response) {
  response.writeHead(200, {'Content-Type': 'text/plain'});
  response.end('Hello World\n');
}).listen(8124);
```

O JavaScript funciona em uma única thread, e o Node emula um ambiente assíncrono em um ambiente com uma única thread por meio de um *laço de eventos* (event loop), com funções de *callback* associadas que são chamadas sempre que um evento específico é disparado. No Exemplo 1.1, quando uma solicitação web é recebida, a função de callback é chamada.

A mensagem do `console.log()` é mostrada no terminal assim que a função criadora do servidor é chamada. O programa não termina aí e fica esperando que uma solicitação web seja feita, bloqueando o terminal

enquanto espera.

```
console.log('Server running at http://127.0.0.1:8124/');
```

Laços de eventos e funções de callback

Veremos mais detalhes sobre os laços de eventos do Node, seu suporte à programação assíncrona e as funções de callback no Capítulo 2.

Uma vez que o servidor foi criado e recebeu uma solicitação, a função de callback envia ao navegador um cabeçalho em texto puro com um status `200`, escreve a mensagem de Hello World e encerra a resposta.

Parabéns! Você criou seu primeiro servidor web no Node com apenas algumas linhas de código. Não é algo particularmente útil, a não ser que seu único interesse seja acenar para o mundo. Ao longo deste livro, aprenderemos a fazer aplicações de maior utilidade usando o Node, mas antes de deixar o Hello World para trás, que tal modificá-lo para que fique mais interessante?

Um Hello World "tunado"

Enviar uma mensagem estática demonstra, em primeiro lugar, que a aplicação funciona, e em segundo lugar, como criar um servidor web simples. Esse exemplo trivial também demonstra alguns elementos-chave de uma aplicação em Node. Contudo, sempre podemos enriquecê-la um pouquinho, deixando-a mais divertida. Pensando nisso, ajustei o código aqui e ali para criar uma segunda aplicação com alguns truques bacanas.

O código mexido está no Exemplo 1.2. Nele modifiquei a aplicação original para analisar a solicitação entrante em busca de uma query string. O nome na string é extraído e usado para determinar o tipo de conteúdo devolvido. Praticamente qualquer nome que usarmos retornará uma resposta personalizada, mas se a query for `name=burningbird` teremos uma imagem. Se nenhuma query string estiver presente, ou se nenhum nome for passado a ela, a variável recebe o valor-padrão '`world`'.

Exemplo 1.2 – Hello World melhorado

```
var http = require('http');
var fs = require('fs');
```

```
http.createServer(function (req, res) {
   var name = require('url').parse(req.url, true).query.name;

   if (name === undefined) name = 'world';

   if (name == 'burningbird') {
      var file = 'phoenix5a.png';
      fs.stat(file, function (err, stat) {
         if (err) {
            console.error(err);
            res.writeHead(200, {'Content-Type': 'text/plain'});
            res.end("Sorry, Burningbird isn't around right now \n");
         } else {
            var img = fs.readFileSync(file);
            res.contentType = 'image/png';
            res.contentLength = stat.size;
            res.end(img, 'binary');
         }
      });
   } else {
      res.writeHead(200, {'Content-Type': 'text/plain'});
      res.end('Hello ' + name + '\n');
   }
}).listen(8124);

console.log('Server running at port 8124/');
```

O resultado de acessar a aplicação web com a query string `?name=burningbird` é mostrado na Figura 1.3.
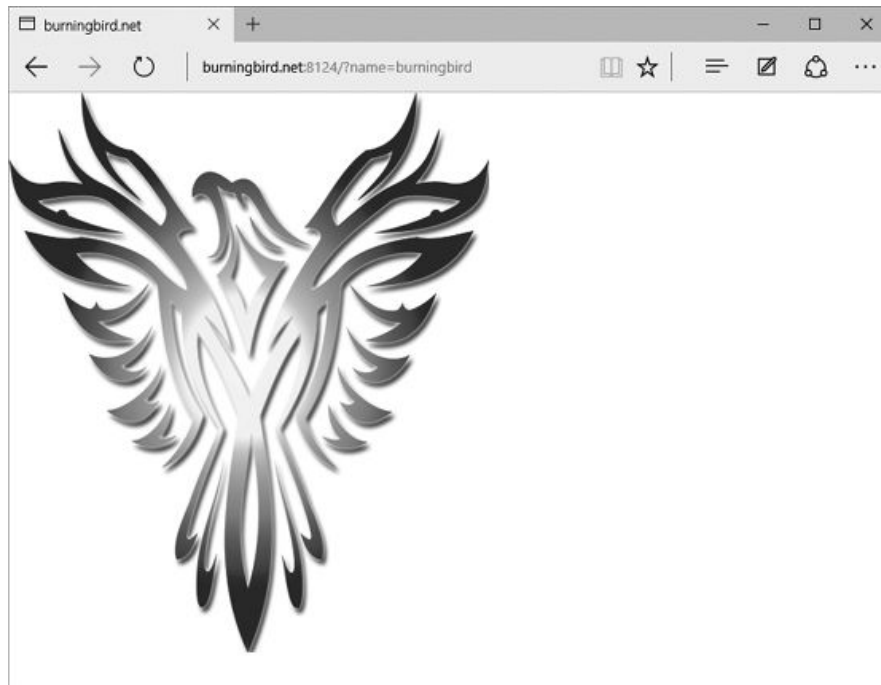
Figura 1.3 – Olá, passarinho!

Não há muito código a mais, mas existem diferenças notáveis entre o Hello World básico e a versão melhorada. Logo no início, um novo módulo é convocado na aplicação, de nome `fs`. É o módulo File System (sistema de arquivos), e nos encontraremos com ele muitas vezes nos próximos capítulos. Existe ainda um terceiro módulo, mas sua chamada é bem diferente das outras duas:

```
var name = require('url').parse(req.url, true).query.name;
```

As propriedades de um módulo podem ser encadeadas, e com isso podemos importá-lo e usar suas funções na mesma linha. Isso acontece com bastante frequência no módulo URL, cuja única finalidade é oferecer recursos utilitários para manipular URLs.

Os nomes dos parâmetros de solicitação e resposta são abreviados, respectivamente, para `req` e `res`, a fim de facilitar seu acesso. Depois de analisar a solicitação para extrair o valor de `name`, a primeira coisa que fazemos é testá-lo para determinar se é `undefined`. Se for, o valor é ajustado para o padrão, `world`. Se existir um valor definido para `name`, é testado novamente para ver se é `burningbird`. Se não for, a resposta é parecida com o que tínhamos na aplicação mais simples, com a exceção de que o nome

informado é repassado à mensagem de retorno.

Se o nome for `burningbird`, entretanto, estamos lidando com uma imagem em vez de um texto. O método `fs.stat()` não apenas verifica se o arquivo existe, como também devolve um objeto com informações sobre o arquivo, incluindo seu tamanho. Esse valor é usado para criar o cabeçalho.

Se o arquivo de imagem não existir, a aplicação lida com a situação com graça e leveza, emitindo uma mensagem informando que o pássaro flamejante foi dar uma voltinha, além de uma mensagem de erro no console, usando o método `console.error()` desta vez:

```
{ [Error: ENOENT: no such file or directory, stat 'phoenix5a.png']
  errno: -2,
  code: 'ENOENT',
  syscall: 'stat',
  path: 'phoenix5a.png' }
```

Se a imagem existir, será lida e atribuída a uma variável, que por sua vez será devolvida na resposta. Os valores do cabeçalho serão reajustados de acordo.

O método `fs.stats()` usa a estrutura-padrão de código[2] para callbacks do Node com o valor do erro no primeiro parâmetro – frequentemente chamado de *errback*. Entretanto, o trecho que lê a imagem pode tê-lo feito coçar a cabeça. Não parece certo, pelo menos não como as outras funções do Node que vimos neste capítulo (e provavelmente na maioria dos exemplos encontrados na internet). A diferença é que estou usando uma função síncrona, `readFileSync()`, em vez da versão assíncrona, `readFile()`.

O Node suporta ambas as versões, síncrona e assíncrona, da maioria das funções para manipulação de sistemas de arquivos encontradas no módulo File System. Geralmente, usar uma operação assíncrona do Node em uma solicitação web é considerado tabu, mas o recurso está lá para quem precisar. A versão assíncrona do mesmo trecho de código é mostrada no Exemplo 1.3.

```
fs.readFile(file, function(err,data) {
              res.contentType = 'image/png';
              res.contentLength = stat.size;
```

```
    res.end(data, 'binary');
});
```

Como escolher qual usar? Em algumas circunstâncias, as operações de leitura e gravação em disco podem não causar impacto no desempenho, independentemente do tipo de função usada. Nesses casos, a função síncrona é mais limpa e fácil de usar. Ela também evita muito código aninhado – um problema típico do sistema de callback do Node, o qual detalharemos no Capítulo 2.

Embora eu não empregue tratamento de exceções neste exemplo, as funções síncronas permitem o uso de `try...catch`. Não podemos usar esse tradicional tratamento de erros com as funções assíncronas (por isso mesmo o valor do erro é o primeiro parâmetro da função anônima de callback).

Entretanto, a lição mais importante a ser aprendida com esse segundo exemplo é que nem todo I/O do Node precisa ser assíncrono.

Buffers, I/O assíncrono e os módulos File System e URL

Estudaremos o módulo URL em detalhes no Capítulo 5 e o módulo File System no Capítulo 6. Contudo, observe que o módulo File System será usado em todos os capítulos do livro. O conceito de buffers e do processamento assíncrono será explicado no Capítulo 2.

Opções de linha de comando do Node

Nas últimas duas seções, o Node foi chamado no terminal sem ativar nenhuma opção. Antes de continuar, gostaria de mostrar algumas dessas opções. Outras mais serão introduzidas ao longo do livro sempre que necessário.

Para descobrir todas as opções e argumentos disponíveis, use a ajuda embutida, grafada como `-h` ou `--help`:

```
$ node --help
```

Isso lista todas as opções disponíveis e mostra a sintaxe correta a seguir quando executamos uma aplicação no Node:

```
Usage: node [options] [ -e script | script.js ] [arguments]
       node debug script.js [arguments]
```

Para saber a versão do Node, use o comando:

```
$ node -v
```

ou

```
$ node --version
```

Para verificar a sintaxe de uma aplicação em Node, use a opção `-c`. Isso verifica a sintaxe do seu código sem que a aplicação precise ser executada:

```
$ node -c script.js
```

ou

```
$ node --check script.js
```

Para descobrir quais são as opções do V8, digite:

```
$ node --v8-options
```

Isso devolve várias opções, incluindo `--harmony`, usada para habilitar por completo os recursos Harmony do JavaScript, o que inclui toda a funcionalidade ES6 que já foi implementada, mas ainda não está disponível por padrão nas versões LTS ou Current.

Uma opção de que gosto muito é `-p` ou `--print`, que pode analisar uma linha de um programa em Node[3] e mostrar o resultado. Ela é especialmente útil quando estamos verificando as propriedades ambientais do processo, que discutiremos melhor no Capítulo 2. Um exemplo é a linha a seguir, que mostra todos os valores da propriedade `process.env`:

```
$ node -p "process.env"
```

Ambientes de hospedagem do Node

À medida que aprendemos sobre o Node, nos contentamos em testá-lo em nosso próprio ambiente local, seja ele Windows, OS X ou Linux. Quando chegar o momento de abrir sua aplicação a um público maior, será preciso encontrar um ambiente para rodar a aplicação em Node, como uma rede virtual privativa (VPN), a alternativa que costumo usar, ou um provedor de hospedagem que ofereça suporte a aplicações Node. O primeiro requer alguma experiência em administrar servidores conectados à internet, enquanto o segundo pode limitar o que nossa aplicação em Node pode ou não fazer.

Hospedando o Node em seu próprio servidor, VPS ou provedor de hospedagem

Hospedar o Node no mesmo servidor que roda seu site no WordPress acabará se mostrando um beco sem saída, por conta das necessidades especiais do Node. Você não precisa, *necessariamente*, ter acesso administrativo ou root no servidor para rodar o Node, mas deveria. Além disso, muitos provedores de hospedagem não ficarão muito felizes caso sua aplicação use várias portas de comunicação, o que pode comprometer dos sistemas da empresa.

Todavia, hospedar o Node em um servidor privativo virtual (Virtual Private Server, ou VPS) é bastante simples, como é o meu caso na Linode, que acesso por VPN. Em um VPS, o assinante tem acesso ao usuário root e, portanto, tem controle total e absoluto sobre a máquina virtual, desde que não coloque em risco os outros usuários que possam estar na mesma máquina física. Muitas empresas que oferecem VPSs garantem que cada conta individual é completamente isolada das demais, e que nenhuma conta consegue "sugar" todos os recursos do sistema compartilhado[4].

Entretanto, o problema com um VPS é semelhante ao que teria se pudesse manter seu próprio servidor físico: é você mesmo quem tem de administrar e manter o servidor. Isso inclui instalar e configurar o sistema de email e o servidor web alternativo, normalmente um Apache ou Nginx (*https://www.nginx.com*), além de lidar com firewalls, segurança, email etc. Não é nada trivial.

Ainda assim, caso se sinta à vontade administrando todos os aspectos de um ambiente conectado à internet, um VPS é uma opção econômica para hospedar uma aplicação em Node – ou, pelo menos, até que ela esteja pronta para entrar em produção, e nesse caso vale a pena considerar hospedá-la em uma *solução na nuvem* (cloud hosting).

Nas nuvens

Hoje em dia, uma aplicação fatalmente acabará residindo em um servidor em nuvem (cloud server) sem muita modificação em relação aos

computadores individuais ou pertencentes a um grupo. As aplicações em Node se dão muito bem em implementações baseadas na nuvem.

Quando uma aplicação em Node está hospedada na nuvem, o que fazemos, na realidade, é criar a aplicação em nosso próprio servidor ou estação de trabalho, testá-la para ter certeza de que faz o que queremos e depois remetê-la ao servidor na nuvem. Um servidor Node em nuvem permite criar a aplicação Node exatamente como queremos, usando os recursos de qualquer banco de dados (ou qualquer outro recurso) que desejemos, mas sem ter de administrar o próprio servidor diretamente. Podemos nos concentrar apenas na aplicação do Node sem nos preocupar com servidores de FTP ou email, ou qualquer tipo de manutenção.

Git e GitHub: pré-requisitos para desenvolvimento em Node

Se você nunca usou o Git, um sistema de controle de código-fonte, será preciso instalá-lo em seus ambientes e aprender como usar. Toda a evolução de código em aplicações Node, incluindo o envio da versão final para o servidor na nuvem, acontece pelo Git.

O Git é um software livre de código aberto, gratuito e fácil de instalar. Podemos acessar o software pelo seu site oficial (*https://git-scm.com*). Há inclusive um guia interativo (*https://try.github.io/levels/1/challenges/1*) que podemos usar para aprender os comandos básicos do Git, hospedado no site GitHub.

Aliás, sempre que há Git, há também GitHub. O código-fonte do próprio Node.js é mantido no GitHub, onde também estão a maioria, senão todos, os módulos do Node. O código-fonte para os exemplos deste livro também estão disponíveis no GitHub.

O GitHub (*https://github.com*) é provavelmente o maior repositório de código-fonte open source do mundo. Definitivamente, é o centro do universo quando se trata de Node. Ele pertence a uma empresa privada, mas está disponível gratuitamente para a maioria dos usuários. A empresa por trás do GitHub oferece excelente documentação (*https://help.github.com/categories/bootcamp/*) sobre como usar o site, e há muitos livros e tutoriais para ajudar a aprender tanto Git quanto GitHub. Dentre eles, podemos destacar um livro gratuito e disponível para download sobre Git (*https://git-scm.com/book/en/v2*), bem como os livros *Version Control with Git* (O'Reilly), de Loeliger e McCullough, e *Introdução ao GitHub* (Novatec), de autoria de Bell e Beer.

O paradigma para hospedar uma aplicação em Node nas nuvens é bastante semelhante em praticamente todos os provedores desse tipo de serviço. Primeiro, crie a aplicação em Node, seja localmente, na sua estação de trabalho, seja em um servidor próprio. Quando estiver pronto para testar em um ambiente semelhante ao de produção, é hora de

começar a procurar um bom provedor de serviços em nuvem. Para a maioria dos que conheço, basta se cadastrar, criar um projeto e especificar que é baseado em Node, caso o serviço seja capaz de hospedar ambientes diferentes. Talvez você precise especificar quais outros recursos serão necessários, como, por exemplo, bancos de dados.

Quando estiver tudo pronto para a implementação, enviamos a aplicação à nuvem. Normalmente, usamos o Git para isso, ou então uma ferramenta fornecida pelo provedor de nuvem. Por exemplo, o ambiente de cloud da Microsoft, o Azure, usa o Git para enviar a aplicação de seu ambiente local para a nuvem, enquanto o Google Cloud Platform fornece sua própria caixa de ferramentas para esse processo.

Encontrando um provedor

Embora não seja tão atualizado, um bom lugar para procurar um provedor de hospedagem para projetos Node é a página do GitHub criada para esse fim (*https://github.com/nodejs/node-v0.x-archive/wiki/Node-Hosting*).

Node: a versão LTS e o processo de atualização

Em 2014, o mundo do Node ficou perplexo (ou, pelo menos, alguns de nós ficamos) quando um grupo de mantenedores do Node se separaram da comunidade, criando uma dissidência do Node.js chamada io.js. A razão para esse desentendimento foi política: o pessoal do io.js considerava que a Joyent, empresa que mantinha o Node, estava vagarosa demais no processo de passar a governança do projeto para a comunidade. O grupo também reclamava que a Joyent estava atrasada em providenciar o suporte para as constantes atualizações do V8, a "máquina" JavaScript por trás do Node.

Felizmente, os dois grupos resolveram suas diferenças e reuniram esforços em um único produto novamente, ainda chamado de Node.js. Agora, o Node é mantido por uma organização não governamental sem fins lucrativos, a Node Foundation, sob os auspícios da Linux Foundation. Como resultado, a base de código de ambos os grupos foi combinada, e, em vez de ganhar a versão 1.0, sua versão inicial estável se tornou o Node 4.0, representando o "devagar e sempre" do Node em direção à versão 1.0 e

a rapidez do io.js, que chegou à versão 3.0 antes da reunião.

A nova semântica de versões do Node

Um resultado dessa união foi a adoção de um calendário bastante rigoroso de lançamentos para o Node, baseado em semântica de versões (semantic versioning, ou Semver, *http://semver.org*), que é bastante conhecida dos usuários do Linux. Cada nova versão lançada pelo modelo Semver tem três grupos de números, cada um com um significado específico. Por exemplo, no momento em que escrevo estas linhas, a versão do Node.js em meu servidor Linux é 4.3.2, que se traduz como:

- A versão (major release, chamada coloquialmente em português de "versão grande") é 4. Esse número só será aumentado quando uma mudança muito grande for feita no Node, tão grande que se torna incompatível com a anterior.
- A revisão (minor release, ou "versão pequena") é 3. Esse número aumenta sempre que novas funcionalidades são adicionadas ao Node, mas ele continua compatível com as revisões anteriores da versão 4.
- A atualização (patch release) é 2. Esse número muda quando uma correção de segurança ou outras correções forem aplicadas, e também é compatível com as versões anteriores.

Estou usando, na minha estação de trabalho Windows, a versão estável (Stable) 5.7.1, e faço testes de código em uma máquina Linux com a versão atual (Current) 6.0.0.

A Node Foundation também suporta um ciclo de lançamentos mais coeso, embora mais problemático, do que o ciclo normal atualize-no-dia-ou-perca-o-bonde a que estamos acostumados. Esse ciclo começou com a primeira versão LTS (Long-Term Support) do Node.js v4, que será suportado até abril de 2018. A Node Foundation lançou sua primeira versão Stable, o Node.js v5, no final de outubro de 2015. O Node 5.x.x teve suporte apenas até abril de 2016, quando foi substituído pelo Node.js v6. A estratégia agora é ter uma versão Current (o nome Stable não existe mais) a cada seis meses, mas apenas as versões pares virarão LTS, como o

Node v4.

Lançamento da versão 6.0.0 como Current

Em abril de 2016, foi lançada a versão 6.0.0 do Node, que substituiu a versão 5.x, e que será transformada na nova LTS em outubro de 2016. A Node Foundation substituiu por Current a antiga designação da versão em desenvolvimento ativo, antigamente chamada de Stable.

Depois de abril de 2018, o Node v4 entrará em modo de manutenção. Nesse meio tempo, haverá novas atualizações compatíveis com versões antigas (conhecidas como *semver-major bumps*), bem como atualizações para correção de bugs e falhas de segurança.

Qual versão o livro usa?

Este livro usa a versão LTS, ou seja, o Node.js v4. Há notas indicando as diferenças entre v4 e v5/v6, sempre que necessário.

Independentemente de qual versão LTS você decida usar, sempre será necessário atualizá-la quando sair alguma correção de segurança ou de bugs. Atualizar a versão "grande" para um novo semver-major bump é algo que apenas você, ou sua organização, pode decidir como fazer. Entretanto, a atualização será sempre compatível com versões anteriores, e o impacto será apenas no funcionamento interno do Node. Ainda assim, é uma boa ideia passar antes por um plano de atualização e testes.

Qual versão você deve usar? Em um ambiente corporativo, o melhor é adotar a versão LTS, que no momento em que escrevo é o Node.js v4. Contudo, se o seu ambiente empresarial puder se adaptar mais rapidamente às mudanças que causam impacto, você pode ter acesso aos recursos mais modernos do V8 e do Node se usar a versão Current mais recente.

O mundo mágico dos ambientes de teste e produção

Estudaremos os procedimentos de teste e depuração do Node, bem como outros processos de desenvolvimento e produção, no Capítulo 11.

Atualizando o Node

Agora que o calendário de novas versões do Node é mais apertado, mantê-lo atualizado é ainda mais crítico. Felizmente, o processo de upgrade é indolor, além de oferecer alternativas.

Podemos verificar a versão atual com o comando:

```
node -v
```

Se estiver usando um instalador de pacotes, basta empregar o procedimento-padrão de atualização. Dessa forma, o Node e outros softwares em seu sistema são atualizados (o comando `sudo` não é necessário no Windows):

```
sudo apt-get update
sudo apt-get upgrade --show-upgraded
```

Se estiver usando outro instalador de pacotes, siga as instruções associadas a ele descritas no site oficial do Node. Negligenciar esse procedimento pode levar a uma falta de sincronia entre o seu Node e a versão mais atual.

Também podemos usar o npm para atualizar o Node, usando a sequência de comandos a seguir:

```
sudo npm cache clean -f
sudo npm install -g
sudo n stable
```

Para instalar a versão mais atual do Node no Windows, OS X ou Raspberry Pi, baixe o instalador a partir da página de downloads do Node.js e execute. A nova versão é instalada por cima da anterior.

Gerenciador de versões do Node

Em ambientes Linux ou OS X, podemos usar o Node Version Manager, cujo comando é nvm (*https://github.com/creationix/nvm*), para manter o Node atualizado.

O Node Package Manager (npm, *https://www.npmjs.com*) é atualizado com mais frequência que o próprio Node. Para atualizar apenas ele, rode o seguinte comando:

```
sudo npm install npm -g n
```

Esse comando instala a versão mais nova da aplicação. Podemos verificar a nova versão pelo comando:

```
npm -v
```

Entretanto, tenha em mente que isso pode causar problemas, especialmente se o ambiente é compartilhado por uma equipe. Se os

membros da equipe estiverem usando a versão do npm que foi instalada com o Node e você atualizar só ele manualmente, podem ocorrer resultados inconsistentes de compilação não muito fáceis de descobrir.

Veremos o npm com mais profundidade no Capítulo 3. Por ora, note que podemos manter todos os módulos do Node atualizados com o comando a seguir:

```
sudo npm update -g
```

Node, V8 e ES6

Por trás do Node existe um "motor" JavaScript. Para a maioria das aplicações, esse motor (em inglês, "engine") é o V8. Originalmente criado pelo Google para o Chrome, o código-fonte do V8 foi aberto ao público em 2008. O V8 foi criado para melhorar o desempenho do JavaScript, incorporando um compilador just-in-time (JIT) que traduz o código em JavaScript para código de máquina, em vez de interpretá-lo. O JavaScript foi, por muitos anos, uma linguagem interpretada. O V8 foi escrito em C++.

Até a Microsoft tem uma dissidência do Node.js

A Microsoft fez uma divisão (fork) do Node para criar uma versão que usa seu próprio engine de JavaScript, chamado Chakra, especialmente para ser usado em sua visão sobre como a Internet das Coisas (IoT) deve ser. Mais sobre esse fork no Capítulo 12.

Quando o Node v4.0 foi lançado, funcionava sobre o V8 na versão 4.5, a mesma versão do engine usado no Chrome. Os mantenedores do Node estão comprometidos a adotar as versões futuras do V8 à medida que forem lançadas, o que significa que o Node, hoje, suporta muitos dos novos recursos do ECMA-262 (ECMAScript 2015 ou, simplesmente, ES6).

Suporte do Node v6 ao V8

O Node v6 suporta o V8 na versão 5.0, e novas versões do Node adotarão as novas versões do V8 correspondentes.

Nas versões mais antigas do Node, para acessar os novos recursos do ES6, era preciso usar a flag harmony (`--harmony`) quando a aplicação fosse executada:

```
node --harmony app.js
```

Hoje, o suporte aos recursos do ES6 é baseado nos seguintes critérios (informação retirada diretamente da documentação do Node.js):

- Todos os recursos de *shipping*, que o V8 considera estáveis, estão ativados por default no Node.js e não precisam de nenhuma flag na hora da execução.
- Recursos *staged*, que são recursos quase finalizados, mas que ainda não são considerados estáveis o suficiente pela equipe do V8, só são ativados pela flag de execução: `--es_staging` (ou seu sinônimo, `--harmony`).
- Recursos *in-progress* podem ser ativados individualmente pela sua respectiva flag harmony (por exemplo, `--harmony_destructuring`), embora isso seja expressamente desencorajado, a não ser para testes.

Veremos o suporte ao ES6 no Node e como usar de forma eficaz os diferentes recursos no Capítulo 9. Por ora, saiba que os recursos ES6 a seguir são *apenas alguns* dos suportados pelo Node, de fábrica:

- Classes
- Promessas
- Símbolos
- Funções-vetor
- Geradores
- Coleções
- `let`
- O operador `spread`

Avançado: os complementos do Node em C/C++

Com o Node instalado, e depois que você teve a chance de brincar um pouco com ele, pode estar se perguntando o que, exatamente, você instalou.

Apesar de a linguagem que usamos para criar aplicações em Node ser o

JavaScript, a maior parte do próprio Node é, na verdade, escrita em C++. Normalmente essa informação é irrelevante para a maioria das aplicações que usamos, mas se você conhecer C/C++ pode estender a funcionalidade do Node com *complementos* (em inglês, add-ons) escritos nessas linguagens.

Escrever um complemento para o Node não é o mesmo que escrever uma aplicação tradicional em C/C++. Para começar, existem bibliotecas, como o próprio V8, que você deverá acessar. Além disso, um add-on do Node não é compilado com as ferramentas que você usaria normalmente.

A documentação do Node para complementos mostra um exemplo Hello World de add-on. Verifique o código do exemplo curto, que deve parecer bastante familiar para quem já programou em C/C++. Uma vez escrito o código, é preciso usar a ferramenta `node-gyp` para compilar o add-on, criando um arquivo *.node*.

Primeiro, é criado um arquivo de configuração chamado *binding.gyp*, que usa um formato semelhante a JSON para oferecer informações sobre o add-on:

```
{
  "targets": [
    {
      "target_name": "addon",
      "sources": [ "hello.cc" ]
    }
  ]
}
```

A configuração do add-on é executada com o seguinte comando:

```
node-gyp configure
```

Isso cria o arquivo de configuração apropriado (um Makefile no Unix, um vcxproj no Windows) e o coloca no diretório *build/*. Para compilar o novo add-on do Node, execute o comando:

```
node-gyp build
```

O add-on compilado é instalado no diretório *build/release* e está agora disponível para uso. Podemos importá-lo para qualquer aplicação da mesma forma que os outros complementos já instalados no Node por

padrão (que veremos no Capítulo 3).

Mantendo módulos nativos do Node

Embora faça parte do escopo deste livro, se você tiver interesse em criar módulos nativos do Node (os add-ons), esteja ciente das diferenças entre plataformas. Por exemplo, a Microsoft tem instruções especiais para módulos nativos no Azure (*https://azure.microsoft.com/en-us/documentation/articles/nodejs-use-node-modules-azure-apps/*), e o mantenedor do popular módulo nativo *node-serialport* escreveu um artigo detalhando as dificuldades por que passou enquanto mantinha esse módulo (*http://www.voodootikigod.com/on-maintaining-a-native-node-module/*).

Obviamente, se você nunca viu C/C++, pode querer criar módulos usando JavaScript mesmo, e veremos como fazer isso no Capítulo 3. Contudo, se conhecer essas duas linguagens, um add-on pode ser bem mais eficaz, especialmente para alguma necessidade específica do sistema.

Um aspecto a considerar e sempre ter em mente é o turbilhão de alterações drásticas pelas quais o Node passou desde a versão 0.8 até a nova versão 6.x. Para conter quaisquer problemas que possam ocorrer, é preciso instalar o NAN, ou Native Abstractions for Node.js (*https://github.com/nodejs/nan*). Esse arquivo de cabeçalho ajuda a suavizar as diferenças entre as versões do Node.js.

---

1 N. do T.: Sendo uma função de um objeto, o nome mais adequado para `createServer` seria método, embora função não esteja errado. Observe também que a autora batizou a variável com o mesmo nome do módulo, embora isso não seja obrigatório, pois funcionaria com qualquer outro nome. Por exemplo, `var servidorWeb = require('http');`. Obviamente, você precisaria substituir também http.`createServer` por `servidoWeb.createServer`.

2 N. do T.: Em português, as palavras standard, default e pattern são traduzidas normalmente como “padrão”. Entretanto, em programação, especialmente em JavaScript, cada uma tem um significado diferente. Pior ainda, são comumente usadas juntas duas a duas, e em algumas situações, as três juntas (“the default standard pattern”, o pesadelo de qualquer tradutor). Por isso, neste livro, standard é traduzido como padrão, não traduzimos default (é sempre default, termo que qualquer programador conhece) e pattern é traduzido como “estrutura-padrão de”. Code pattern, portanto, ficaria “estrutura-padrão de código”, e design pattern ficaria “estrutura-padrão de projeto”. A outra tradução usual para pattern, “modelo”, também gera confusão com os modelos de dados (data models), por isso não foi usada.

3 N. do T.: Dizer que um programa ou script está escrito “em Node” não é completamente correto, mas é o tipo de simplificação tolerada tanto em inglês quanto em português. Como já deve estar claro neste ponto, o Node é um framework para trazer o JavaScript (uma linguagem notoriamente “de navegador”) para o servidor. Não se programa “em Node”, propriamente, mas sim “em JavaScript, sobre o framework Node”. Entretanto, coloquialmente é comum ouvir os programadores falarem dessa forma, e a própria autora, que tem um estilo de texto bastante leve e descontraído, usa assim.

4 N. do T.: Em um VPS, cada assinante possui sua própria máquina virtual, mas várias dessas máquinas virtuais compartilham a mesma máquina física.

CAPÍTULO 2

Blocos de construção do Node: objetos globais, eventos e sua natureza assíncrona

Embora ambos sejam construídos em JavaScript, o ambiente de aplicações que rodam no navegador é muito diferente do ambiente de servidor. Uma diferença fundamental entre o Node e seu primo que mora no navegador é o *buffer* para dados binários. O Node, agora, tem acesso aos tipos *ArrayBuffer* e *typed arrays* do ES6. Todavia, a maior parte da funcionalidade de dados binários no Node é implementada na classe `Buffer`.

O `buffer` é um dos objetos globais do Node. Outro objeto global seria o próprio `global`, embora o objeto global no Node seja fundamentalmente diferente do objeto global a que estamos acostumados no navegador. Os desenvolvedores do Node têm acesso a outro objeto global, `process`, que constrói uma ponte entre a aplicação em Node e seu ambiente.

Felizmente, um aspecto do Node deve ser bastante familiar aos desenvolvedores de frontend: sua natureza assíncrona orientada a eventos. A diferença no Node é que estamos vigiando a abertura de arquivos e não botões que podem ser clicados pelo usuário.

Ser orientado a eventos também significa que nossas velhas amigas, as funções de temporização (timer), estão à nossa disposição no Node.

Os módulos e o console

Falarei sobre outros componentes globais – `require`, `exports`, `module` e `console` – mais adiante. Veremos `require`, `exports` e `module` no Capítulo 3 e `console` no Capítulo 4.

Objetos global e process

Dois objetos fundamentais em Node são `global` e `process`. O objeto `global` é semelhante ao objeto global no navegador, com algumas diferenças bastante significativas. O objeto `process`, entretanto, é totalmente Node até a última ponta.

Objeto global

No navegador, quando declaramos uma variável no nível mais alto da hierarquia de classes, ela é declarada com escopo global. No Node, as coisas não funcionam assim. Ao declarar uma variável em um módulo ou aplicação, não obtemos uma variável disponível globalmente, mas sim restrita ao módulo ou aplicação em que foi declarada. Portanto, podemos tranquilamente criar uma variável "global" chamada `str` em um módulo e outra de mesmo nome na aplicação, e não acontecerá nenhum conflito.

Para demostrar a diferença, criaremos uma função simples que soma um número fornecido como parâmetro a outro predeterminado (chamado de base) e retorna o resultado. A função em JavaScript criada por nós será usada em dois lugares: em uma biblioteca chamada por uma página HTML e como módulo em uma aplicação Node.

O código para a biblioteca JavaScript, gravada em um arquivo chamado *add2.js*, declara uma variável `base`, atribui a ela o valor 2 e a adiciona a qualquer número que tenha sido passado à função:

```
var base = 2;

function addtwo(input) {
   return parseInt(input) + base;
}
```

O módulo é quase idêntico, muito simples e faz basicamente a mesma coisa, com algumas pequenas diferenças decorrentes da sintaxe de módulos do Node. No Capítulo 3 veremos o objeto `module` em detalhes. Por ora, copie o código a seguir em um arquivo chamado *addtwo.js*:

```
var base = 2;

exports.addtwo = function(input) {
  return parseInt(input) + base;
};
```

Para demonstrar as diferenças de comportamento do escopo `global` em ambos os ambientes, criamos dois chamadores. A biblioteca *add2.js* é usada em uma página HTML, que também declara uma variável `base`:

```
<!DOCTYPE html>
<html>
   <head>
      <script src="add2.js"></script>
      <script>

         var base = 10;
         console.log(addtwo(10));
      </script>
   </head>
<body>

</body>
</html>
```

Ao acessar a página, o console do navegador mostra o valor 20, em vez do 12 que seria o esperado. A razão é que todas as variáveis criadas do lado de fora de uma função JavaScript no navegador são adicionadas ao mesmo objeto global. Quando declaramos uma nova variável chamada `base` na página, substituímos com ela o valor atribuído dentro do arquivo de script.

Agora, vamos experimentar a mesma coisa com o módulo `addtwo` em uma aplicação Node:

```
var addtwo = require('./addtwo').addtwo;

var base = 10;

console.log(addtwo(base));
```

O resultado na aplicação Node foi 12. Veja que declaramos uma nova variável `base` na aplicação, mas esta não perturbou de nenhuma forma o valor de `base` no módulo, porque ambos existem em namespaces globais diferentes.

Poder usar namespaces distintos, em vez de um único compartilhado, é um belo aprimoramento, embora não seja universal. Há algumas coisas que o `global` ainda compartilha em todos os ambientes: acesso aos objetos e funções do Node disponíveis globalmente, incluindo o objeto `process`,

que veremos adiante. Podemos confirmar o fato adicionando a linha a seguir a um arquivo e rodando-o como uma aplicação Node. Os objetos e funções disponíveis globalmente são mostrados no console:

```
console.log(global);
```

Objeto process

O objeto `process` é um componente essencial no ambiente do Node, pois oferece informação sobre o ambiente de execução. Além disso, os processos de entrada e saída-padrão (I/O) ocorrem por meio do `process`; é também por ele que podemos encerrar de forma apropriada uma aplicação e até sinalizar quando o *laço de eventos* (event loop) do Node chegou ao fim de um ciclo. Veremos o laço de eventos mais adiante, na seção "Fila de eventos (loop)".

O objeto `process` é usado em muitas aplicações no decorrer do livro. Por ora, veremos mais de perto a funcionalidade de informações sobre o ambiente do objeto `process`, bem como a importantíssima entrada e saída-padrão.

O objeto `process` dá acesso a informações sobre o ambiente do Node, bem como do ambiente de execução. Para explorá-lo, usaremos a opção `-p` quando chamarmos o `node` no terminal, que executa um script e retorna um resultado imediato. Por exemplo, veja a propriedade `process.versions` no seguinte exemplo:

```
$ node -p "process.versions"
{ http_parser: '2.5.0',
  node: '4.2.1',
  v8: '4.5.103.35',
  uv: '1.7.5',
  zlib: '1.2.8',
  ares: '1.10.1-DEV',
  icu: '56.1',
  modules: '46',
  openssl: '1.0.2d' }
```

Aspas simples ou duplas: quando usar no terminal?

Observe que usamos aspas duplas, que são obrigatórias na janela de comandos do Windows. Como as aspas duplas funcionam em qualquer ambiente, use-as em todos os

seus scripts sempre.

Foram listadas as versões de vários componentes e dependências do Node, incluindo o V8, OpenSSL (a biblioteca para comunicações seguras), o próprio Node, e muitos mais.

A propriedade `process.env` oferece um número expressivo de informações sobre o que o Node vê em seu ambiente de desenvolvimento ou produção:

```
$ node -p "process.env"
```

São de particular interesse as diferenças entre arquiteturas, como, por exemplo, Linux *versus* Windows.

Para explorar os valores de `process.release`, use o comando a seguir:

```
$ node -p "process.release"
```

O que obteremos depende do que estiver instalado. Tanto no LTS quanto no Current, teremos o nome da aplicação, bem como URLs para o código-fonte. Contudo, em LTS temos uma propriedade adicional:

```
$ node -p "process.release.lts"
'Argon'
```

Se acessarmos o mesmo valor na versão Current, como a v6, teremos um resultado diferente:

```
$ node -p "process.release.lts"
undefined
```

As informações ambientais são uma maneira de o desenvolvedor entender o que o Node vê antes e durante o desenvolvimento. Contudo, não inclui as dependências da maioria dos dados usados diretamente em sua aplicação porque, como vimos, podem não ser consistentes entre duas versões distintas do Node. Mesmo assim, pare a leitura agora e dedique um tempo a explorar os dados apresentados pelos comandos.

O que deve ser consistente em todas as versões do Node são inúmeros objetos e funções essenciais para um grande número de aplicações. Dentre elas temos o acesso ao I/O padrão e a possibilidade de encerrar de forma apropriada uma aplicação.

Os fluxos-padrão são canais de comunicação preestabelecidos entre uma

aplicação e seu ambiente. Eles consistem de uma entrada-padrão (`stdin`), uma saída-padrão (`stdout`) e uma saída de erros padrão (`stderr`). Em uma aplicação Node, esses canais oferecem comunicação direta entre a aplicação em Node e o terminal.

O Node suporta canais com três funções de processo:

- `process.stdin` – um fluxo de leitura para `stdin`
- `process.stdout` – um fluxo de escrita para `stdout`
- `process.stderr` – um fluxo de escrita para `stderr`

Não conseguimos fechar esses fluxos, ou encerrá-los dentro da aplicação, mas podemos receber dados vindos do canal `stdin` e escrever nos canais `stdout` e `stderr` para enviar dados para fora da aplicação.

As funções de I/O de `process` herdam de `EventEmitter`, que exploraremos na seção "EventEmitter", o que significa que elas podem emitir eventos, que podem ser capturados pelo seu código e processados. Para processar os dados usando `process.stdin`, primeiramente precisamos determinar a codificação usada no fluxo. Se não o fizermos, os resultados serão empacotados em um tipo buffer, em vez do tipo string:

```
process.stdin.setEncoding('utf8');
```

Depois, precisamos monitorar o evento `readable`, que informa a existência de um conjunto de dados pronto para ser lido. Usaremos a função `process.stdin.read()` para ler esses dados. Caso os dados de entrada não sejam `null`, vamos ecoá-los para fora do processo via `process.stdout`, usando a função `process.stdout.write()`:

```
process.stdin.on('readable', function() {
   var input = process.stdin.read();

   if (input !== null) {
      // ecoa na saída o texto de entrada
      process.stdout.write(input);
   }
});
```

Poderíamos deixar de determinar a codificação do texto e obter os mesmos resultados – estaríamos lendo um buffer na entrada e escrevendo um buffer na saída –, mas, para o usuário da aplicação, pareceria que

estamos trabalhando com texto simples (strings). Só que não estamos. A próxima função de `process` que exploraremos demonstra essa diferença.

No Capítulo 1, criamos um servidor web muito simples, que monitorava a existência de solicitações HTTP e respondia com uma mensagem. Para finalizar o programa, seria preciso matar o processo com o sinal apropriado (kill) ou pressionar Ctrl-C. Em vez disso, podemos encerrar a aplicação de dentro dela própria usando `process.exit()`. Podemos até mesmo sinalizar se a aplicação foi encerrada com sucesso ou se ocorreu alguma falha.

Modificaremos nossa aplicação simples de I/O para monitorar a ocorrência de uma string de saída e, quando ela ocorrer, encerrar o programa. O Exemplo 2.1 mostra a aplicação completa.

Exemplo 2.1 – Demonstrando o I/O padrão no Node e encerrando a aplicação

```
process.stdin.setEncoding('utf8');
process.stdin.on('readable', function() {
   var input = process.stdin.read();

   if (input !== null) {
      // ecoa saída o texto de entrada
      process.stdout.write(input);

      var command = input.trim();
      if (command == 'exit')
         process.exit(0);
   }
});
```

Quando executamos a aplicação, qualquer string que digitamos é imediatamente transferida para a saída. Entretanto, se digitarmos **`exit`**, a aplicação é encerrada sem que precisemos recorrer ao Ctrl-C.

Se removermos a chamada à função `process.stdin.setEncoding()` no início do código, a aplicação falhará. A razão é simples: não existe uma função `trim()` em um buffer. Podemos converter localmente o buffer para string e, depois, aplicar o `trim`:

```
var command = input.toString().trim();
```

Mas a melhor prática mesmo é definir a codificação para todo o

aplicativo, de forma global, e assim evitar quaisquer efeitos colaterais indesejados.

A interface Stream

Os objetos de I/O do `process` são implementações da interface Stream, abordada em detalhes no Capítulo 6.

O objeto `process.stderr` faz exatamente o que o nome diz: podemos escrever nele quando um erro ocorre. Por que usaríamos ele em vez de `process.stdout`? Pelo mesmo motivo pelo qual o canal `stderr` foi criado: para haver uma diferenciação entre a saída esperada e as mensagens de saída que indicam quando um problema ocorreu. Em alguns sistemas, podemos até processar `stderr` de forma diferente do que é produzido por `stdout` (por exemplo, as mensagens que saem por `stdout` podem ser redirecionadas para um arquivo de logs, enquanto `stderr` pode ir diretamente para o console).

Existem dezenas de outros objetos e funções úteis associadas ao `process`, e veremos muitos deles ao longo do livro.

Buffers, typed arrays e strings

No JavaScript executado no navegador, nos velhos tempos, nunca houve a necessidade de lidar com o fluxo de dados binários. Originalmente, o JavaScript foi idealizado para trabalhar com valores em string passados por campos em um formulário ou enviados para janelas de alerta. Mesmo quando o Ajax surgiu para perturbar a ordem natural das coisas, os dados que fluíam entre cliente e servidor continuaram sendo strings e codificados em Unicode.

Contudo, o cenário mudou radicalmente no momento em que as exigências impostas ao JavaScript começaram a ficar mais sofisticadas. Em paralelo com o Ajax, temos também WebSockets, e mesmo o que podemos fazer no próprio navegador é muito mais abrangente hoje – em lugar do simples acesso a formulários, temos tecnologias avançadas como WebGL e Canvas.

A solução no JavaScript e no navegador foi usar um `ArrayBuffer`,

manipulado por meio de *vetores tipados*, ou *typed arrays*. Em Node, a solução é o `Buffer`.

Originalmente, os dois conceitos não eram a mesma coisa. Entretanto, quando o io.js e o Node.js se uniram para formar o Node v4.0.0, o Node acabou ganhando o suporte aos vetores tipados do V8 versão 4.5. O buffer do Node é, agora, sustentado por um typed array chamado `Uint8Array`, que representa um vetor de elementos inteiros de 8 bits sem sinal. Entretanto, isso não significa que podemos usar um em lugar do outro. No Node, a classe `Buffer` é a estrutura primária de dados usada na maioria das operações de I/O e não podemos substituí-la diretamente por um typed array; fazê-lo provocaria um erro. Além disso, converter um buffer do Node em um typed array pode até ser factível, mas sempre serão introduzidas algumas falhas. De acordo com a documentação da API do `Buffer`, quando "convertemos" um buffer em um typed array:

- A memória do buffer é copiada, não compartilhada.
- A memória do buffer é interpretada como um array comum, não como um array de bytes. Em outras palavras, `new Uint32Array(new Buffer([1,2,3,4]))` cria um array de quatro elementos do tipo `Uint32Array`, cujos elementos são `[1,2,3,4]`, e não um array `Uint32Array` com um único elemento `[0x1020304]` ou `[0x4030201]`.

Portanto, podemos usar ambos os tipos de manipulação de fluxo binário no Node, mas, no geral, emprega-se preferencialmente o buffer. Então, o que é um buffer para o Node?

O que é um fluxo binário?

O octeto é uma unidade de medida na computação, sendo composto por valores de 8 bits – daí o nome. Em um sistema que suporta bytes de 8 bits, um octeto e um byte são a mesma coisa. Um fluxo (stream) é apenas uma sequência de dados. Portanto, um arquivo binário é, no fim das contas, uma sequência de octetos.

Um buffer no Node é um bloco de dados binários brutos que foram alocados fora da pilha interna do V8. É gerenciado pela classe `Buffer`. Uma vez alocado, o buffer não pode ser redimensionado.

O buffer é o tipo de dados default para acesso a arquivos: a não ser que uma codificação específica seja fornecida quando ler e escrever em um

arquivo, os dados são inseridos ou extraídos de buffers.

No Node v4, podemos criar um buffer diretamente usando a palavra-chave `new`:

```
let buf = new Buffer(24);
```

Tenha sempre em mente que, ao contrário de `ArrayBuffer`, ao criar um buffer no Node seu conteúdo não é inicializado. Para garantir que não existirão dados peculiares e possivelmente sigilosos no buffer, o que pode levar a resultados inesperados e desastrosos, é uma boa ideia preencher o buffer imediatamente após criá-lo:

```
let buf = new Buffer(24);
buf.fill(0); // preenche todo o buffer com zeros
```

Podemos também preenchê-lo parcialmente, especificando as posições de início e fim.

Especificando a codificação do preenchimento de buffer

A partir do Node v5.7.0, podemos também especificar a codificação a ser usada para o `buf.fill()`, com a sintaxe: `buf.fill(string[, início[, fim]] [, codificação])`.

Podemos criar diretamente o novo buffer passando para a função construtora um array de octetos, outro buffer ou uma string. O buffer é criado com o conteúdo copiado de qualquer um dos três. Para a string, se não estiver em UTF-8, é preciso especificar a codificação; as strings no Node são codificadas em UTF-8 (`utf8` ou `utf-8`) por default.

```
let str = 'New String';
let buf = new Buffer(str);
```

Não quero descrever cada método existente na classe `Buffer`, visto que o Node oferece farta documentação a respeito, mas quis mostrar mais de perto algumas de suas funcionalidades.

Diferenças entre o Node v4 e o Node v5/v6

Os tipos de codificação `raw` e `raws` foram removidos nas versões v5 e posteriores do Node.

Contudo, no Node v6 os construtores foram descontinuados em favor de novos métodos em `Buffer`: `Buffer.from()`, `Buffer.alloc()` e `Buffer.allocUnsafe()`.

Com a função `Buffer.from()`, fornecer um array retorna um buffer com uma cópia de seu conteúdo. Todavia, se passarmos um `ArrayBuffer`, informando

opcionalmente seu tamanho e offset em bytes, o buffer compartilha a mesma memória que um `ArrayBuffer`. Passar um buffer copia os elementos do buffer e passar uma string copia os elementos da string.

A função `Buffer.alloc()` cria um buffer preenchido de um tamanho determinado, enquanto `Buffer.allocUnsafe()` cria um buffer de tamanho determinado, mas que pode conter dados antigos ou sigilosos presentes anteriormente na memória e que devem ser sobrescritos com `buf.fill()` para garantir que sejam apagados.

O código a seguir, em Node:

```
'use strict';
let a = [1,2,3];
let b = Buffer.from(a);
console.log(b);
let a2 = new Uint8Array([1,2,3]);
let b2 = Buffer.from(a2);
console.log(b2);
let b3 = Buffer.alloc(10);
console.log(b3);
let b4 = Buffer.allocUnsafe(10);
console.log(b4);
```

resulta nos seguintes buffers em meu sistema:

```
<Buffer 01 02 03>
<Buffer 01 02 03>
<Buffer 00 00 00 00 00 00 00 00 00 00>
<Buffer a0 64 a3 03 00 00 00 00 01 00>
```

Observe os dados aleatórios resultantes do uso de `Buffer.allocUnsafe()`, comparado a `Buffer.alloc()`.

Buffer, JSON, StringDecoder e strings em UTF-8

Os buffers podem ser convertidos em objetos JSON e em strings. Para demonstrar, digite o seguinte código em um arquivo Node e o execute na linha de comando:

```
"use strict";

let buf = new Buffer('This is my pretty example');
let json = JSON.stringify(buf);

console.log(json);
```

O resultado é:

```
{"type":"Buffer",
"data":[84,104,105,115,32,105,115,32,109,121,32,112,114,101,116,
116,121,32,101,120,97,109,112,108,101]}
```

O JSON especifica que o tipo de objeto sendo transformado é um `Buffer` e que logo em seguida estão os dados. Obviamente, o que estamos vendo são os dados depois de serem armazenados em um buffer como uma sequência de octetos, um formato ilegível para nós, humanos.

ES6

A maioria dos exemplos de código usam estruturas bastante familiares em JavaScript, conhecidas por todos há anos. Entretanto, costumo de tempos em tempos dar uma espiada no ES6. Veremos a relação entre o Node e o ES6 (EcmaScript 2015) em detalhes no Capítulo 9.

Para fechar o ciclo, podemos extrair de dentro do objeto JSON os dados brutos do buffer e depois usar o método `Buffer.toString()` para convertê-los em uma string, como mostrado no Exemplo 2.2.

Exemplo 2.2 – De string a buffer, de buffer a JSON, voltando para buffer e, por fim, para string

```
"use strict";

let buf = new Buffer('This is my pretty example');
let json = JSON.stringify(buf);

let buf2 = new Buffer(JSON.parse(json).data);

console.log(buf2.toString()); // meu exemplo bonitinho
```

A função `console.log()` mostra a string original depois que ela foi recriada a partir dos dados do buffer. A função `toString()` converte a string para UTF-8 por default, mas se quiséssemos outros tipos de string bastaria passar a codificação correta:

```
console.log(buf2.toString('ascii')); // meu exemplo bonitinho
```

Podemos especificar a posição de início e fim na conversão de strings:

```
console.log(buf2.toString('utf8', 11,17)); // bonitinho
```

Usar `buffer.toString()` não é a única maneira de converter um buffer em uma string. Podemos também usar uma classe auxiliar, `StringDecoder`. O único objetivo desse objeto é decodificar valores de buffer para strings em UTF-8, mas faz isso com um pouco mais de flexibilidade e recuperabilidade. Se o método `buffer.toString()` receber uma sequência incompleta de caracteres UTF-8, retornará lixo incompreensível. Por outro lado, a função `StringDecoder` preenche a sequência fragmentada até que esteja completa e devolve o resultado. Se os dados a serem processados estiverem em UTF-8 e vierem de um fluxo do qual recebemos blocos em vez de um fluxo contínuo, é obrigatório usar `StringDecoder`.

Um exemplo das diferenças entre as diversas rotinas de conversão de strings pode ser conferido na aplicação Node a seguir. O símbolo de euro (€) é codificado como três octetos, mas o primeiro buffer contém apenas os primeiros dois octetos; o segundo buffer contém o terceiro.

```
"use strict";

let StringDecoder = require('string_decoder').StringDecoder;
let decoder = new StringDecoder('utf8');

let euro = new Buffer([0xE2, 0x82]);
let euro2 = new Buffer([0xAC]);

console.log(decoder.write(euro));
console.log(decoder.write(euro2));

console.log(euro.toString());
console.log(euro2.toString());
```

O resultado no console é uma linha em branco e uma segunda linha com o símbolo de euro (€), quando usamos `StringDecoder`, mas duas linhas de sujeira caótica quando usamos `buffer.toString()`.

Podemos também converter uma string em um buffer preexistente usando `buffer.write()`. Entretanto, é importante que o buffer tenha o tamanho correto para abrigar a quantidade de octetos necessária à representação de todos os caracteres. Lembre-se, o símbolo do euro precisa de três octetos para representá-lo (0xE2, 0x82, 0xAC):

```
let buf = new Buffer(3);
buf.write('€','utf-8');
```

Isso é também uma bela demonstração do fato de que o número de caracteres UTF-8 não é equivalente ao número de octetos em um buffer. Em caso de dúvida, sempre podemos verificar o tamanho do buffer com `buffer.length`:

```
console.log(buf.length); // 3
```

Manipulação de buffers

Já conseguimos ler e escrever conteúdo em um buffer, em qualquer posição dentro dele (o chamado offset), empregando diversas funções tipadas. Exemplos de uso dessas funções aparecem no trecho de código a seguir, que escreve quatro inteiros de 8 bits sem sinal em um buffer e depois os lê e exibe no console:

```
var buf = new Buffer(4);

// escreve os valores no buffer
buf.writeUInt8(0x63,0);
buf.writeUInt8(0x61,1);
buf.writeUInt8(0x74,2);
buf.writeUInt8(0x73,3);

// mostra o buffer em formato string
console.log(buf.toString());
```

Tente você mesmo, copiando o código em um arquivo e o executando. Podemos ainda ler individualmente cada inteiro de 8 bits usando `buffer.readUInt8()`.

O Node suporta a leitura e escrita de inteiros de 8, 16 e 32 bits, com ou sem sinal, bem como números em ponto flutuante de precisão simples (float) e dupla (double). Para todos os tipos que não sejam inteiros de 8 bits, podemos escolher ainda se queremos o formato *little-endian* ou *big-endian* de ordenação de bytes. A seguir, alguns exemplos de funções que levam em conta a ordenação:

- `buffer.readUIntLE()` – Lê o valor em uma determinada posição no buffer usando little-endian.
- `buffer.writeUInt16BE()` – Escreve um inteiro de 16 bits sem sinal em uma posição do buffer usando big-endian.

- `buffer.readFloatLE()` – Lê um valor em ponto flutuante em uma posição do buffer usando little-endian.
- `buffer.writeDoubleBE()` – Escreve um valor de ponto flutuante em precisão dupla (64 bits) usando big-endian.

Ordenação de bytes

Se você não estiver familiarizado com a ordenação de bytes em uma palavra de dados, ou endianness em inglês (ou, ainda, byte order), basta saber que é o formato que determina como um valor de dois ou mais bytes é armazenado. O formato escolhido determina se o byte mais significativo é armazenado na posição mais baixa de memória (big-endian) ou se o byte menos significativo é armazenado na posição mais baixa de memória (little-endian).

A Figura 2.1, que extraí da Wikipedia, mostra visualmente a diferença entre ambos.
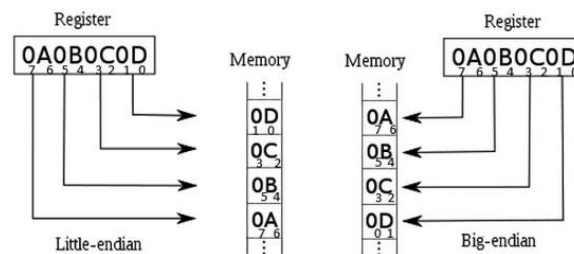
Figura 2.1 – Demonstração dos formatos little-endian e big-endian. Imagem gentilmente cedida pela Wikipedia.

Podemos ainda escrever inteiros de 8 bits diretamente usando um formato que lembra o de um array:

```
var buf = new Buffer(4);

buf[0] = 0x63;
buf[1] = 0x61;
buf[2] = 0x74;
buf[3] = 0x73;
```

Além de ler e escrever em uma posição (offset) específica dentro do buffer, podemos também criar um novo buffer que consiste em uma seção do antigo, usando `buffer.slice()`. Esse recurso é particularmente interessante porque modificar o conteúdo do novo buffer também modifica o conteúdo do antigo. O Exemplo 2.3 demonstra esse comportamento ao criar um buffer a partir de uma string, depois fatiando o buffer existente para criar outro e modificando o conteúdo do novo buffer. Ambos os buffers são impressos no console para que possamos perceber como essa

modificação *in-place* funciona.

Exemplo 2.3 – Demonstrando uma alteração in-place no buffer antigo quando o novo buffer é modificado

```
var buf1 = new Buffer('this is the way we build our buffer');
var lnth = buf1.length;

// cria um novo buffer como uma fatia do antigo
var buf2 = buf1.slice(19,lnth);
console.log(buf2.toString()); // mostra o buffer novo

//modifica o segundo buffer
buf2.fill('*',0,5);
console.log(buf2.toString()); // ***** nosso buffer

// mostra impacto no primeiro buffer
console.log(buf1.toString()); // foi desta maneira que ******* nosso buffer
original
```

Se você precisar testar a equivalência de dois buffers, pode usar a função `buffer.equals()`:

```
if (buf1.equals(buf2)) console.log('buffers are equal');
```

Podemos também copiar os bytes de um buffer para o outro usando `buffer.copy()`. Podemos copiar todos ou parte dos bytes usando parâmetros opcionais. Contudo, observe que se o segundo buffer não for grande o bastante para armazenar todo o conteúdo, este será truncado e teremos apenas a porção de dados que couberam no buffer de destino:

```
var buf1 = new Buffer('this is a new buffer with a string');

// copia o buffer
var buf2 = new Buffer(10);
buf1.copy(buf2);

console.log(buf2.toString()); // apenas "this is a"
```

Para comparar buffers, usamos `buffer.compare()`, que devolve um valor indicando se o buffer comparado vem antes do outro, em ordem alfabética. Se vier antes, o valor devolvido é -1; se vier depois, o valor é 1. Se os dois buffers tiverem bytes equivalentes, o valor 0 é devolvido:

```
var buf1 = new Buffer('1 is number one');
var buf2 = new Buffer('2 is number two');

var buf3 = new Buffer(buf1.length);
```

```
buf1.copy(buf3);

console.log(buf1.compare(buf2)); // -1
console.log(buf2.compare(buf1)); // 1
console.log(buf1.compare(buf3)); // 0
```

Há ainda outra classe de buffer, `SlowBuffer`, que pode ser usada em qualquer situação na qual for preciso reter o conteúdo de um buffer pequeno por um longo período de tempo. Normalmente, o Node cria buffers a partir de um bloco de memória pré-alocado se o buffer for pequeno (tamanho menor que 4 kB). Dessa forma, o coletor de lixo não precisa rastrear e limpar os inúmeros blocos pequenos de memória que acabam se formando ao longo do tempo.

A classe `SlowBuffer` permite criar buffers pequenos fora dessa memória pré-alocada, fazendo com que persistam por períodos mais longos de tempo. Contudo, é fácil imaginar que o uso dessa classe pode provocar um impacto significativo no desempenho, e deve ser usado apenas quando *nada* mais funcionar.

Gerenciamento de callbacks e eventos assíncronos no Node

O JavaScript é single-threaded, o que o torna inerentemente síncrono. Isso significa que o JavaScript é executado, linha a linha, até que a aplicação seja finalizada. Como o Node é baseado em JavaScript, herda deste seu comportamento síncrono de uma única thread.

Entretanto, se for necessária alguma funcionalidade que implique esperar pela ocorrência de alguma coisa, como, por exemplo, a abertura de um arquivo, uma resposta a uma solicitação web ou outra atividade dessa natureza, nossa aplicação ficará bloqueada (travada) até que a operação esteja terminada. Isso é um ponto de falha de bastante impacto em uma aplicação cliente-servidor.

A solução para evitar o bloqueio é o event loop, ou laço de eventos.

Fila de eventos (loop)

Para que possamos ter funcionalidade assíncrona, as aplicações podem adotar um dos dois caminhos a seguir. O primeiro seria criar uma thread

para cada processo que consuma tempo. O resto do código poderia ser distribuído dessa forma, em paralelo. O problema com essa solução é que as threads são “caras”, consumindo muitos recursos da máquina e gerando grande complexidade na aplicação.

O segundo caminho é adotar uma arquitetura baseada em eventos. Quando um processo que consome tempo é invocado, a aplicação não espera que ele termina. Em vez disso, o processo sinaliza quando já tiver terminado pela emissão de um sinal de evento. Esse evento é adicionado à *fila de eventos*, ou *event loop*. Qualquer funcionalidade dependente desse evento registra seu interesse nele, e quando o evento é finalmente retirado da fila e processado, a funcionalidade dependente é chamada, e os dados relacionados ao evento são passados a ela.

Tanto o JavaScript no navegador quanto o no Node empregam o segundo caminho. No navegador, quando adicionamos um manipulador de cliques de mouse (`click` handler) a um elemento, o que você na verdade fez foi se “cadastrar” (ou assinar) um evento e fornecer uma função de callback, que será chamada quando o evento acontecer, liberando o resto da aplicação para que continue:

```
<div id="someid"> </div>
<script>
   document.getElementById("someid").addEventListener("click",
                                          function(event) {
       event.target.innerHTML = "I been clicked!";
   }, false);
</script>
```

O Node tem seu próprio laço de eventos, mas em vez de esperar por um evento de UI, como o clique do mouse em algum elemento da página, esse laço é usado para facilitar funcionalidades de servidor, em sua maioria de entrada e saída (I/O). Isso inclui eventos associados a abertura de arquivos, sinalizando que o arquivo foi aberto, lendo seu conteúdo, colocando-o em um buffer e notificando o cliente quando o processo foi finalizado. Outro exemplo são as solicitações web vindas de um usuário. Esses tipos de processo não apenas consomem uma grande quantidade de tempo como também se utilizam de uma grande quantidade de recursos

do sistema. Cada acesso ao recurso tipicamente trava esse recurso, impedindo que outro processo o acesse até que o processo original encerre. Além disso, aplicações web dependem da ação do usuário e, às vezes, de outras aplicações.

O Node processa todos os eventos da fila em ordem. Quando chegar ao evento em que você está interessado, chama a função de callback informada e passa a ela qualquer informação associada ao evento.

No servidor web básico que criamos como primeiro exemplo no Capítulo 1, vimos o laço de eventos em ação. Repetirei o código aqui para facilitar a análise:

```
var http = require('http');

http.createServer(function (request, response) {
  response.writeHead(200, {'Content-Type': 'text/plain'});
  response.end('Hello World\n');
}).listen(8124);

console.log('Server running at http://127.0.0.1:8124/');
```

No Exemplo 2.4, modifiquei o código para separar as ações individuais e também para monitorar mais alguns eventos que ocorrem durante a criação do servidor, a conexão do cliente e o processo de "escuta" do servidor.

Exemplo 2.4 – Servidor web básico com destaque para alguns eventos

```
var http = require('http');

var server = http.createServer();

server.on('request', function (request, response) {

   console.log('request event');

   response.writeHead(200, {'Content-Type': 'text/plain'});
   response.end('Hello World\n');
});

server.on('connection', function() {
   console.log('connection event');
});

server.listen(8124, function() {

   console.log('listening event');
});
```

```
console.log('Server running on port 8124');
```

Note que a função `requestListener()`, o callback de solicitação do servidor, não é mais chamada pela função `http.createServer()`. Em vez disso, a aplicação atribuiu a uma variável o servidor HTTP que acabou de ser criado e a usa para capturar dois eventos:

- O evento `request`, emitido sempre que uma solicitação web chega, vinda de um cliente
- O evento `connection`, emitido cada vez que um cliente se conecta à aplicação web

Em ambos os casos, é possível "se inscrever" nesses eventos usando a função `on()`, que a classe HTTP herda de `EventEmitter`. Mais adiante discutiremos a respeito do objeto que implementa essa funcionalidade herdada por HTTP, mas, por ora, vamos nos concentrar nos eventos. Há ainda outro evento que recebeu inscrição, o evento de escuta, que é acessado pela função de callback passada à função `server.listen()`.

Um objeto, o servidor HTTP e três eventos: `request`, `connection` e `listening`. Então, o que acontece quando a aplicação é criada e as solicitações web são feitas?

Ao iniciar a aplicação, imediatamente é mostrada no console uma mensagem informando que o "Servidor está rodando na porta 8124". Isso acontece porque a aplicação não fica bloqueada quando o servidor é criado, ou quando um cliente se conecta, ou quando começamos a monitorar as solicitações entrantes. Portanto, a primeira mensagem `console.log()` é mostrada somente depois que todas as funções assíncronas não blocantes já foram executadas.

A próxima mensagem depois dessa é "listening event" (evento de escuta). Assim que criamos o servidor, queremos monitorar a entrada de novas conexões e solicitações. Fazemos isso ao chamar a função `server.listen()`. Não precisamos esperar por nenhum evento avisando que "o servidor terminou de ser criado", pois a função `http.createServer()` retorna imediatamente o controle ao programa principal. Podemos testar esse comportamento inserindo uma mensagem `console.log()` diretamente após a chamada a `http.createServer()`. Ao ser adicionada, ela será a primeira a

ser mostrada no console quando a aplicação for iniciada.

Na versão anterior da aplicação, a função `server.listen()` está encadeada com a função `http.createServer()`, mas não precisa ser assim. Está dessa forma por conveniência e por elegância de código, não por necessidade da orientação a eventos. Entretanto, a função `server.listen()` é uma função assíncrona com callback, que é chamado quando o evento de escuta é emitido. Portanto, a mensagem de console é emitida *depois* da mensagem que informa que o servidor está na porta 8124.

Nenhuma outra mensagem é mostrada até que um cliente se conecte à aplicação. Nesse momento, recebemos a mensagem do evento de conexão, porque a conexão é o primeiro evento invocado por um novo cliente. Depois disso temos uma ou duas mensagens de evento de solicitação. A razão para essa diferença é a forma como cada navegador faz uma solicitação a um novo site. O Chrome quer o recurso especificado, mas também pede o *favicon.ico*, por isso a aplicação recebe duas solicitações. O Firefox e o IE não se importam com o favicon, portanto a aplicação só recebe uma mensagem de solicitação desses navegadores.

Se recarregarmos a página no mesmo navegador, só obteremos a mensagem do evento de solicitação. A conexão já está estabelecida e é mantida até que o usuário feche o navegador ou algum timeout ocorra. Acessar o mesmo recurso por navegadores diferentes estabelece uma conexão separada para cada um.

Acessar a aplicação web usando o Chrome resulta nas seguintes mensagens no console:

- Server running on port 8124
- Listening event
- Connection event
- Request event
- Request event

Se tivermos uma função em um módulo ou diretamente na aplicação que queiramos tornar assíncrona, precisamos defini-la usando um critério específico, mostrado a seguir.

Criando uma função assíncrona de callback

Para demonstrar a estrutura fundamental da funcionalidade de callback, a aplicação completa mostrada no Exemplo 2.5 cria um objeto com uma única função, `doSomething()` (em português, faça alguma coisa). A função recebe três argumentos: o primeiro é devolvido como dado se não houver erros, o segundo deve ser uma string e o terceiro é a função de callback. Em `doSomething()`, se o segundo argumento estiver faltando ou não for uma string, o objeto cria um novo objeto `Error`, que é devolvido na função de callback. Se nenhum erro ocorrer, a função de callback é chamada, o erro se torna `null` e os dados da função são retornados (neste caso, o primeiro argumento).

Os elementos-chave da funcionalidade de callback estão em negrito no Exemplo 2.5.

Exemplo 2.5 – Estrutura fundamental da última funcionalidade de callback

```
var fib = function (n) {
    if (n < 2) return n;
    return fib(n - 1) + fib(n - 2);
};

var Obj = function() { };

Obj.prototype.doSomething = function(arg1_) {
   var callback_ = arguments[arguments.length - 1];
   callback = (typeof(callback_) == 'function' ? callback_ : null);
   var arg1 = typeof arg1_ === 'number' ? arg1_ : null;
   if (!arg1)
      return callback(new Error('first arg missing or not a number'));

      process.nextTick(function() {

         // bloqueia CPU
         var data = fib(arg1);
         callback(null, data);
   });
}

var test = new Obj();
var number = 10;

test.doSomething(number, function(err,value) {
      if (err)
```

```
        console.error(err);
    else
        console.log('fibonaci value for %d is %d', number, value);
});

console.log('called doSomething');
```

A primeira funcionalidade-chave é garantir que o último argumento seja uma função de callback e que o primeiro argumento da função de callback seja um erro. Como observado no Capítulo 1, esse padrão de código é frequentemente chamado de *errback*. Não podemos adivinhar a intenção do usuário, mas podemos garantir que o último argumento seja uma função, e isso tem que ser o bastante.

A segunda funcionalidade-chave é a criação de um objeto `Error` adicional no Node e sua devolução como resposta à função de callback. Não podemos empregar `throw...catch` aqui porque estamos no mundo assíncrono, portanto o tratamento de erros deve necessariamente ser feito no objeto `Error` do callback.

A última funcionalidade crítica é a chamada à função de callback, passando a ela os dados da função chamadora caso nenhum erro ocorra. Entretanto, para garantir que este callback seja assíncrono, ele é chamado dentro de uma função `process.nextTick()`. A razão para isso é que `process.nextTick()` garante que o laço de eventos seja esvaziado antes que a função seja chamada. Isso significa que todas as funcionalidades síncronas são processadas antes que a funcionalidade blocante (se existir) seja chamada. No exemplo, o bloqueio não ocorre por algum I/O, mas porque a operação requer muito tempo de CPU. Chamar uma função que calcula a sequência de Fibonacci a partir do valor 10 não toma muito tempo, mas a mesma função chamada com o valor 50 precisará de muito mais tempo, dependendo dos recursos do sistema. A função de Fibonacci é chamada de dentro de `process.nextTick()`, garantindo assim que a funcionalidade consumidora de grandes recursos de CPU seja tratada de forma assíncrona.

Resumindo, tudo o mais pode ser alterado e reorganizado, desde que estas quatro funcionalidades-chave estejam presentes:

- Garanta que o último argumento seja uma função de callback.

- Crie um objeto `Error` do Node e, se um erro ocorrer, devolva esse objeto como primeiro argumento da função de callback.
- Se não ocorrerem erros, chame a função de callback, atribua o valor `null` ao argumento de erro e passe ao callback quaisquer dados relevantes.
- A função de callback deve ser chamada de dentro da função `process.nextTick()` para garantir que não haja bloqueio.

Se alterarmos o valor de `number` para 10, a aplicação mostrará as seguintes mensagens no console:

```
called doSomething
[Error: first argument missing or not a number]
```

Se olharmos o código dos arquivos presentes no diretório *lib* da instalação do Node, veremos que esse padrão de código da função de callback se repete bastante. Embora a funcionalidade possa mudar, o padrão de código é o mesmo.

Essa maneira de fazer as coisas é bastante simples e garante resultados consistentes em métodos assíncronos.

Aninhamento de callbacks

Usar funções de callback é simples, mas gera seus próprios desafios, incluindo o aninhamento de callbacks (às vezes em níveis excessivos). Falaremos sobre o aninhamento excessivo e as soluções para isso no Capítulo 3, na seção "Async, um módulo para gerenciamento eficiente de callbacks".

Anteriormente, mencionei que o objeto `http.Server` herda a funcionalidade de outro objeto e é dele que obtém a capacidade de emitir eventos. Esse objeto é chamado, de forma apropriada, de `EventEmitter`, e falaremos dele a seguir.

O Node é single-thread... na maioria das vezes

O laço de eventos do Node é single-thread. Entretanto, isso não significa que não haja outras threads trabalhando arduamente em segundo plano.

O Node chama algumas de suas funcionalidades, como, por exemplo, as de sistema de arquivos (módulo File System, ou fs), que é implementado em C++ em vez de JavaScript. A funcionalidade de fs usa threads de trabalho para se desincumbir de suas tarefas. Além disso, o Node utiliza a biblioteca libuv, que contém um grupo de threads de trabalho para funcionalidades diversas. A quantidade delas depende do sistema operacional em uso.

Quando nos mantemos fiéis ao JavaScript, criando módulos apenas nessa linguagem, não precisamos nos preocupar com threads de trabalho nem com o libuv. Fato: paternalmente, passaram a mão na nossa cabecinha e nos disseram para não preocupá-la com threads de trabalho. Para mim, isso está ótimo: já trabalhei com ambientes multi-thread e sei a dor de cabeça que eles são.

Entretanto, se estiver interessado em desenvolver extensões Add-on para o Node, vai precisar estar bastante familiarizado com o libuv. Um bom lugar para começar é o artigo *An Introduction to libuv* (*https://nikhilm.github.io/uvbook/basics.html*).

Para saber mais sobre o fascinante mundo perdido do Node multi-thread, sugiro a leitura (também em inglês) das respostas para esta pergunta feita no site Stack Overflow: When is thread pool used?; *http://stackoverflow.com/questions/22644328/when-is-the-thread-pool-used*).

EventEmitter

Basta arranhar um pouco a superfície de muitos dos objetos nativos do Node que encontraremos, por baixo, o `EventEmitter`. Toda vez que virmos um objeto emitir um evento usando a palavra-chave `emit` e logo em seguida um evento tratado pela função `on`, estaremos presenciando o `EventEmitter` em ação. Entender como o `EventEmitter` funciona e como usá-lo são dois dos mais importantes componentes no desenvolvimento em Node.

O `EventEmitter` permite tratar eventos assíncronos no Node. Para demonstrar sua funcionalidade principal, usaremos uma aplicação de teste bem simples.

Primeiro, inclua o módulo `Events`:

```
var events = require('events');
```

Depois, crie uma instância de `EventEmitter`:

```
var em = new events.EventEmitter();
```

Use o `EventEmitter` que acabou de criar para executar duas tarefas essenciais: anexar o manipulador de evento (event handler) a um evento e depois emitir o evento propriamente dito. O manipulador `EventEmitter.on()` é chamado quando um evento específico é emitido. O primeiro parâmetro do método é o nome do evento; o segundo é a função de callback que executará alguma funcionalidade:

```
em.on('someevent', function(data) { ... });
```

O evento é emitido no objeto pelo método `EventEmitter.emit()` quando alguma condição for satisfeita:

```
if (somecriteria) {
   en.emit('data');
}
```

No Exemplo 2.6, criamos uma instância de `EventEmitter` que emite um evento temporizado a cada três segundos. Na função do manipulador deste evento, uma mensagem com um contador é enviada ao console. Observe a correlação entre o argumento `counter` na função `EventEmitter.emit()` e os dados correspondentes na função `EventEmitter.on()` que processa o evento.

Exemplo 2.6 – Um teste bastante simples da funcionalidade EventEmitter

```
var eventEmitter = require('events').EventEmitter;
var counter = 0;

var em = new eventEmitter();

setInterval(function() { em.emit('timed', counter++); }, 3000);

em.on('timed', function(data) {
  console.log('timed ' + data);
});
```

Rodar a aplicação mostra mensagens temporizadas de eventos no console até que a aplicação seja encerrada. O principal ponto a observar nesta aplicação simples é que um evento é disparado pela função `EventEmitter.emit()` e a função `EventEmitter.on()` pode ser usada para capturar esse evento e processá-lo.

Esse exemplo é muito interessante, mas não é particularmente útil. No mundo real, precisamos da funcionalidade de `EventEmitter` em nossos objetos preexistentes, em vez de usar instâncias de `EventEmitter` ao longo da aplicação. É isso o que fazem o `http.Server` e muitas outras classes do Node que compreendem eventos.

A funcionalidade de `EventEmitter` é herdada, portanto precisamos usar outro objeto do Node, `Util`, para ativar essa herança. O módulo `Util` é importado em uma aplicação usando:

```
var util = require('util');
```

Esse módulo é extremamente útil. Veremos a maior parte de sua funcionalidade no Capítulo 11, quando discutiremos sobre a depuração de erros em aplicações Node. Contudo, uma de suas funções é essencial já: `util.inherits()`.

A função `util.inherits()` ativa um construtor para herdar os protótipos de métodos de outra função, um *superconstrutor* (superconstructor). Para tornar `util.inherits()` ainda mais especial, podemos acessar o superconstrutor diretamente nas funções do construtor. Portanto, `util.inherits()` permite que herdemos uma fila de eventos do Node vinda de qualquer classe, e isso inclui `EventEmitter`:

```
util.inherits(Someobj, EventEmitter);
```

Usando `util.inherits()` com um objeto, podemos chamar o método `emit` dentro dos métodos do objeto e definir manipuladores de eventos dentro das instâncias desses objetos:

```
Someobj.prototype.someMethod = function() { this.emit('event'); };
...
Someobjinstance.on('event', function() { });
```

Em vez de tentar decifrar como o `EventEmitter` funciona na teoria, vamos tomar o caminho mais prático do Exemplo 2.7, que mostra uma classe herdando a funcionalidade de `EventEmitter`. Na aplicação, uma nova classe, `inputChecker`, é criada. O construtor pede dois valores, o nome de uma pessoa e um arquivo. O nome da pessoa é atribuído a uma propriedade e um fluxo de escrita é criado usando o método `createWriteStream` do módulo File System.

O objeto também tem um método, `check`, que verifica os dados entrantes em busca de comandos específicos. Um desses comandos (`wr:`) emite um evento de escrita, enquanto `en:` emite um comando de encerramento. Se nenhum comando estiver presente, um evento `echo` é emitido. A instância do objeto cria manipuladores para todos os três eventos: grava dados no arquivo com o evento de escrita, ecoa a entrada na saída quando os dados não contiverem comandos e encerra a aplicação quando recebe um evento desses, usando o método `process.exit`.

Todos os dados de entrada vêm da entrada-padrão (`process.stdin`). A saída gravada usa um fluxo de escrita, que é uma forma de criar uma entidade de saída em segundo plano e na qual as escritas futuras ficam esperando em fila. É um método de gravação em arquivos muito eficiente quando o ambiente impõe uma atividade intensa, que é o caso desta aplicação. Os dados de entrada que acabam sendo ecoados são simplesmente enviados à saída por `process.stdout`.

Exemplo 2.7 – Um objeto orientado a eventos que herda de EventEmitter

```
"use strict";

var util = require('util');
var eventEmitter = require('events').EventEmitter;
var fs = require('fs');

function InputChecker (name, file) {
   this.name = name;
   this.writeStream = fs.createWriteStream('./' + file + '.txt',
      {'flags' : 'a',
       'encoding' : 'utf8',
       'mode' : 0o666});
};

util.inherits(InputChecker,eventEmitter);

InputChecker.prototype.check = function check(input) {

   // elimina excesso de espaços em branco
   let command = input.trim().substr(0,3);

   // processa os possíveis comandos
   // se for wr: grava os dados no arquivo
   if (command == 'wr:') {
      this.emit('write',input.substr(3,input.length));

   // se for en: encerra o processo
   } else if (command == 'en:') {
      this.emit('end');

   // ecoa a entrada na saída-padrão caso não haja comandos
   } else {
      this.emit('echo',input);
   }
};

// testa o novo objeto e o tratamento dos eventos
let ic = new InputChecker('Shelley','output');
```

```
ic.on('write', function(data) {
   this.writeStream.write(data, 'utf8');
});

ic.on('echo', function( data) {
   process.stdout.write(ic.name + ' wrote ' + data);
});

ic.on('end', function() {
   process.exit();
});

// captura a entrada depois de definir a codificação de texto
process.stdin.setEncoding('utf8');
process.stdin.on('readable', function() {
   let input = process.stdin.read();
   if (input !== null)
      ic.check(input);
});
```

Observe que a funcionalidade também inclui o método `process.stdin.on` do manipulador de evento, pois `process.stdin` é um dos muitos objetos do Node que herdam de `EventEmitter`.

Proibido literais octais em modo strict

No Exemplo 2.7, usei o modo *strict* porque empreguei a instrução `let`, exclusiva do ES6. Entretanto, com o modo strict é proibido usar os literais octais (como, por exemplo, 0666) nas permissões de arquivo do descritor de escrita. Em vez disso, é necessário usar a notação `0o666`, que é um literal no estilo do ES6.

A função `on()` é na verdade um atalho para a função `EventEmitter.addListener`, que toma os mesmos parâmetros. Portanto, isto:

```
ic.addListener('echo', function( data) {
   console.log(this.name + ' wrote ' + data);
});
```

É um equivalente direto disto:

```
ic.on('echo', function( data) {
   console.log(this.name + ' wrote ' + data);
});
```

Podemos "ouvir" o próximo evento com `EventEmitter.once()`:

```
ic.once(event, function);
```

Quando temos mais de dez detectores (em inglês, listeners) para um

mesmo evento, recebemos um alerta por default. Para aumentar o número de listeners, use `setMaxListeners`, passando como parâmetro o novo número máximo. O valor zero (`0`) significa que não há limite.

Para remover listeners, use `EventEmitter.removeListener()`:

```
ic.on('echo', callback);
```

```
ic.removeListener('echo',callback);
```

Isso remove um dos listeners presentes no array de detectores de evento, mantendo a ordem. Entretanto, se por qualquer motivo o array de listeners de eventos foi copiado usando `EventEmitter.listeners()`, será necessário recriá-lo após a primeira remoção de um listener.

O laço de eventos do Node e os temporizadores

No navegador temos `setTimeout()` e `setInterval()` para uso com temporizadores e podemos empregar essas mesmas funções no Node. Contudo, elas não são idênticas, pois o navegador usa um laço de eventos mantido pelo próprio navegador e o Node se baseia no laço de eventos de uma biblioteca específica em C++, a já citada libuv – mesmo assim, as diferenças podem ser ignoradas na maioria dos casos.

A versão de `setTimeout()` presente no Node recebe uma função de callback como primeiro parâmetro, o tempo de espera (em milissegundos) no segundo e uma lista opcional de argumentos:

```
setTimeout(function(name) {
              console.log('Hello ' + name);
            }, 3000, 'Shelley');
console.log("waiting on timer...");
```

O nome na lista de argumentos é passado como um argumento para a função de callback definida dentro de `setTimeout()`. O valor da temporização é de 3.000 milissegundos (ou seja, 3 segundos). A mensagem do `console.log()` significa, em português, "esperando o temporizador...", e é mostrada no console imediatamente, pois a função `setTimeout()` é assíncrona.

Podemos cancelar uma temporização se esta for atribuída a uma variável no momento de sua criação. O exemplo a seguir modifica a aplicação

anterior para incorporar um cancelamento rápido de espera e uma mensagem:

```
var timer1 = setTimeout(function(name) {
             console.log('Hello ' + name);
           }, 30000, 'Shelley');

console.log("waiting on timer...");

setTimeout(function(timer) {
            clearTimeout(timer);
            console.log('cleared timer');
           }, 3000, timer1);
```

O temporizador foi definido com um período de tempo razoavelmente longo, o suficiente para que o novo temporizador possa chamar o callback que cancela o temporizador original.

A função `setInterval()` define um intervalador e opera de forma semelhante a `setTimeout()`, com a exceção de que a temporização é reiniciada continuamente até que a aplicação seja encerrada, ou que o temporizador seja esvaziado pela função `clearInterval()`. Se modificarmos o exemplo de `setTimeout()` anterior para incluir uma demonstração de `setInterval()`, a mensagem se repetirá nove vezes antes de ser cancelada.

```
var interval = setInterval(function(name) {
             console.log('Hello ' + name);
           }, 3000, 'Shelley');

setTimeout(function(interval) {
            clearInterval(interval);
            console.log('cleared timer');
           }, 30000, interval);

console.log('waiting on first interval...');
```

Como está devidamente indicado na documentação do Node, não há garantia de que a função de callback seja chamada em exatamente *n* milissegundos (qualquer que seja o *n*). Também é assim com o `setTimeout()` no navegador – não temos controle absoluto sobre o ambiente e alguns fatores podem atrasar levemente o temporizador. Para a maioria das aplicações, a discrepância de tempo é imperceptível, mas, caso estejamos criando animações, o impacto pode ser bastante pronunciado.

Existem duas funções não específicas do Node que podemos usar em

conjunto com os objetos timer e interval e que são devolvidas sempre que `setTimeout()` ou `setInterval()` são chamadas: `ref()` e `unref()`. Se chamarmos `unref()` em um temporizador e ele for o único evento em uma fila de eventos, o temporizador é cancelado e o programa tem permissão para encerrar. Se chamarmos `ref()` no mesmo objeto temporizador, o programa continua rodando até que a temporização tenha sido processada.

Voltando ao primeiro exemplo, criaremos um temporizador mais longo e chamaremos `unref()` para ver o que acontece:

```
var timer = setTimeout(function(name) {
             console.log('Hello ' + name);
           }, 30000, 'Shelley');

timer.unref();

console.log("waiting on timer...");
```

Ao rodar a aplicação, a mensagem é impressa no console e o programa encerra imediatamente. A razão para isso é que o temporizador (timer) definido com `setTimeout()` é o único evento na fila de eventos da aplicação. E se incluirmos outro evento? No código modificado a seguir, adicionamos um intervalador em companhia do temporizador existente e chamamos `unref()` no temporizador:

```
var interval = setInterval(function(name) {
             console.log('Hello ' + name);
           }, 3000, 'Shelley');

var timer = setTimeout(function(interval) {
            clearInterval(interval);
            console.log('cleared timer');
           }, 30000, interval);

timer.unref();

console.log('waiting on first interval...');
```

O temporizador é mantido em funcionamento, o que significa que ele encerra o intervalador. Entretanto, foram os eventos do intervalador que mantiveram o temporizador funcionando por tempo suficiente para permitir que ele esvaziasse o intervalador.

O último grupo de funções de temporização do Node são exclusivos

deste: `setImmediate()` e `clearImmediate()`. `setImmediate()` cria um evento, mas o evento tem precedência sobre os criados por `setTimeout()` e `setInterval()`. Entretanto, não tem precedência sobre eventos de I/O. Além disso, `setImmediate()` não tem um temporizador associado a ele. O evento `setImmediate()` é emitido depois de todos os eventos de I/O, antes de todos os eventos de temporização e na fila de eventos atual. Se o chamarmos de dentro de uma função de callback, ele é colocado no próximo laço de eventos, depois que o laço que o gerou for encerrado. É uma maneira de adicionar um evento ao laço de eventos atual ou ao próximo sem ter que criar temporizadores arbitrários. É mais eficiente que `setTimeout(callback, 0)`, pois tem precedência sobre todos os eventos de temporização.

Há outra função bastante semelhante chamada `process.nextTick()`, mas a função de callback de `process.nextTick()` é chamada depois do encerramento do laço de eventos, porém sempre antes que novos eventos de I/O sejam adicionados. Como demonstrado na seção "Gerenciamento de callbacks e eventos assíncronos no Node", `process.nextTick()` é amplamente usada para implementar a funcionalidade assíncrona no Node.

Callbacks aninhados e tratamento de exceções

Não é nada incomum encontrar uma estrutura como esta em uma aplicação JavaScript que rode no navegador:

```
val1 = callFunctionA();
val2 = callFunctionB(val1);
val3 = callFunctionC(val2);
```

As funções são chamadas em sequência, recebendo os resultados da função anterior e repassando os seus para a próxima. Como todas são síncronas, não precisamos nos preocupar com a possibilidade de as chamadas às funções acabarem ficando fora de ordem – não existem resultados inesperados aqui.

O Exemplo 2.8 mostra um caso muito comum desse tipo de programação sequencial. A aplicação usa as versões síncronas dos métodos do módulo File System para abrir um arquivo e obter seus dados, modificando-os a

fim de substituir todas as ocorrências de “apple” por “orange” e gravar a string resultante em um novo arquivo.

Exemplo 2.8 – Uma aplicação sequencial síncrona

```
var fs = require('fs');

try {
   var data = fs.readFileSync('./apples.txt','utf8');
   console.log(data);
   var adjData = data.replace(/[A|a]pple/g,'orange');

   fs.writeFileSync('./oranges.txt', adjData);
} catch(err) {
  console.error(err);
}
```

Como sempre podem ocorrer problemas e não podemos ter certeza de que os erros serão tratados internamente em uma dada função de um módulo, envolvemos todas as chamadas de função em blocos `try` para tratarmos os erros de forma mais elegante – ou, pelo menos, mais informativa. O exemplo a seguir mostra um erro gerado quando a aplicação não encontra o arquivo a ser lido:

```
{ [Error: ENOENT: no such file or directory, open './apples.txt']
  errno: -2,
  code: 'ENOENT',
  syscall: 'open',
  path: './apples.txt' }
```

Embora não seja muito amigável ao usuário, pelo menos é melhor que a alternativa:

```
$ node nested2
fs.js:549
  return binding.open(pathModule._makeLong(path), stringToFlags(flags), mode);
                 ^
Error: ENOENT: no such file or directory, open './apples.txt'
    at Error (native)
    at Object.fs.openSync (fs.js:549:18)
    at Object.fs.readFileSync (fs.js:397:15)
    at Object.<anonymous>
             (/home/examples/public_html/learnnode2/nested2.js:3:18)
    at Module._compile (module.js:435:26)
    at Object.Module._extensions..js (module.js:442:10)
```

```
    at Module.load (module.js:356:32)
    at Function.Module._load (module.js:311:12)
    at Function.Module.runMain (module.js:467:10)
    at startup (node.js:136:18)
```

Converter o padrão de código de uma aplicação sequencial síncrona em uma implementação assíncrona requer um punhado de modificações. Primeiro, precisamos substituir todas as funções pelas suas versões assíncronas. Entretanto, precisamos ter em mente que cada uma das funções assíncronas não bloqueia o processamento quando chamada, o que significa que não podemos garantir a sequência correta delas porque são chamadas de forma independente umas das outras. A única maneira de assegurar que cada função seja chamada na sequência correta é implementando-as em *callbacks aninhados* (em inglês, nested callbacks).

O Exemplo 2.9 é a versão assíncrona da aplicação do Exemplo 2.8. Todas as funções do módulo File System foram substituídas pela sua versão assíncrona, e as funções são chamadas na sequência correta porque estão em um callback aninhado. Além disso, os blocos `try...catch` foram removidos.

Não podemos usar `try...catch` porque ele seria processado antes que qualquer função assíncrona fosse chamada. Tentar levantar um erro na função de callback seria tentar levantar um erro fora do processo que deveria capturá-lo. Em vez disso, temos de processar o erro diretamente: se o erro aparecer, trate-o e devolva; se não houver erro, continue processando a função de callback.

Exemplo 2.9 – Aplicação do Exemplo 2.8 convertida em assíncrona com o uso de callbacks aninhados

```
var fs = require('fs');
fs.readFile('./apples.txt','utf8', function(err,data) {
   if (err) {
       console.error(err);
   } else {
     var adjData = data.replace(/apple/g,'orange');
     fs.writeFile('./oranges.txt', adjData, function(err) {
        if (err) console.error(err);
```

```
    });
  }
});
```

No Exemplo 2.9, o arquivo de entrada é aberto e lido. Somente quando as duas ações forem finalizadas, a função de callback é chamada. Nessa função, verifica-se se o erro contém algum valor. Em caso afirmativo, o objeto de erro é mostrado no console. Se não houver erros, os dados são processados e o método assíncrono `writeFile()` é chamado. Sua função de callback tem apenas um argumento, o objeto de erro. Qualquer coisa diferente de `null` nesse objeto também será enviada como uma mensagem ao console.

Se um erro ocorrer, será parecido com isto:

```
{ [Error: ENOENT: no such file or directory, open './apples.txt']
  errno: -2,
  code: 'ENOENT',
  syscall: 'open',
  path: './apples.txt' }
```

Se precisar do rastreamento da pilha de erros (stack trace) do erro em questão, basta enviar ao console a propriedade `stack` do objeto de erro do Node:

```
if (err) {
    console.error(err.stack);
}
```

O resultado se pareceria com o seguinte:

```
Error: ENOENT: no such file or directory, open './apples.txt'
    at Error (native)
```

Se precisarmos chamar mais uma função assíncrona ainda respeitando a ordem sequencial, introduziremos mais um nível de aninhamento de callback e, com ele, novos desafios em potencial para o tratamento de erros. No Exemplo 2.10, acessamos uma lista dos arquivos em um diretório. Em cada um dos arquivos, substituímos um nome de domínio genérico por um específico usando o método `replace` de substituição de strings, e o resultado é gravado *no arquivo original*. É mantido um log registrando cada arquivo alterado, usando para isso um fluxo de escrita aberto.

Exemplo 2.10 – Obtendo a listagem do diretório com os arquivos a modificar

```
var fs = require('fs');
var writeStream = fs.createWriteStream('./log.txt',
      {'flags' : 'a',
       'encoding' : 'utf8',
       'mode' : 0666});

writeStream.on('open', function() {
   // obtém lista de arquivos
   fs.readdir('./data/', function(err, files) {

      // para cada arquivo
      if (err) {
         console.log(err.message);
      } else {
         files.forEach(function(name) {

            // modifica conteúdo
            fs.readFile('./data/' + name,'utf8', function(err,data) {

               if (err){
                  console.error(err.message);
               } else {
                  var adjData = data.replace(/somecompany/.com/g,
                             'burningbird.net');

                  // grava no arquivo
                  fs.writeFile('./data/' + name, adjData, function(err)
                    {

                     if (err) {
                        console.error(err.message);
                     } else {

                         // grava no log
                         writeStream.write('changed ' + name + '\n'\n',
                         'utf8', function(err) {

                            if(err) console.error(err.message);
                         });
                     }
                   });
                }
             });
         });
      }
    });
```

```
});

writeStream.on('error', function(err) {
  console.error("ERROR:" + err);
});
```

Primeiro, vemos algo novo: o uso do manipulador de eventos para manipular erros quando chamamos a função `fs.createWriteStream`. A razão para usar o manipulador de eventos é que `createWriteStream` é assíncrona, portanto não podemos usar o tradicional `try...catch`. Ao mesmo tempo, ela não oferece uma função de callback na qual podemos capturar erros. Em vez disso, monitoramos a existência de um evento do tipo `error` e, à guisa de tratamento, o enviamos ao console. Em seguida, procuramos um evento aberto (`open`, ou seja, uma operação que finalizou com sucesso) e executamos o processamento dos arquivos.

A própria aplicação envia o erro diretamente para o console.

Embora a aplicação pareça processar cada arquivo individualmente antes de passar para o próximo, lembre-se de que cada um dos métodos usados na aplicação é assíncrono. Se a aplicação for executada diversas vezes e verificarmos o arquivo *log.txt*, veremos que os arquivos são processados em ordem aparentemente aleatória. Em meu subdiretório *data* havia cinco arquivos. Executar a aplicação três vezes seguidas resultou na seguinte saída para *log.txt* (inseri algumas linhas em branco para maior clareza):

```
changed data1.txt
changed data2.txt
changed data3.txt
changed data4.txt
changed data5.txt

changed data2.txt
changed data4.txt
changed data3.txt
changed data1.txt
changed data5.txt

changed data1.txt
changed data2.txt
changed data5.txt
changed data3.txt
changed data4.txt
```

Outro problema surge caso precisemos verificar o momento em que todos os arquivos terminaram de ser modificados, e assim poder passar a outra tarefa qualquer. O método `forEach` chama de forma assíncrona os iteradores nas funções de callback, portanto eles não bloqueiam o processamento. Adicionar logo após cada `forEach` uma instrução como esta:

```
console.log('all done');
```

não sinaliza de forma alguma que a aplicação foi encerrada, apenas que o `forEach` não bloqueou sua execução. Se adicionarmos uma instrução `console.log` ao mesmo tempo:

```
// escreve no log
writeStream.write('changed ' + name + '\n',
'utf8', function(err) {
    if(err) {
      console.log(err.message);
    } else {
      console.log('finished ' + name);
    }
});
```

e adicionarmos após cada `forEach`:

```
console.log('all finished');
```

teremos, no final, este resultado no console:

```
all finished
finished data3.txt
finished data1.txt
finished data5.txt
finished data2.txt
finished data4.txt
```

Para escapar desse beco sem saída, use um contador que se incrementa a cada mensagem de log e que verifica se seu valor é igual ao comprimento do array de arquivos. Quando os dois valores forem iguais, a mensagem "all done" é finalmente mostrada no console:

```
// antes de acessar o diretório
var counter = 0;
...
  // escrita no log
```

```
writeStream.write('changed ' + name + '\n'\n',
'utf8', function(err) {
   if(err) {
     console.log(err.message);
   } else {
      console.log ('finished ' + name);
      counter++;
      if (counter >= files.length) {
         console.log('all done');
      }
   }
});
```

Agora sim temos o resultado esperado: uma mensagem “all done” é mostrada somente depois que todos os arquivos foram atualizados.

A aplicação funciona razoavelmente bem, mas se perde caso o diretório que estamos processando tenha subdiretórios e arquivos. Se a aplicação encontrar um subdiretório, cospe no console o erro a seguir, embora continue processando o resto da lista:

```
EISDIR: illegal operation on a directory, read
```

O Exemplo 2.11 evita esse tipo de erro usando o método `fs.stats` para retornar um objeto contendo os dados do comando `stat`, que é um comando do sistema operacional Unix. Esse objeto contém informações sobre o elemento, incluindo se é ou não um arquivo. O método `fs.stats` é, obviamente, outro método assíncrono, requerendo mais um nível de aninhamento de callback.

Exemplo 2.11 – Adicionando verificação a cada objeto do diretório para garantir que é um arquivo

```
var fs = require('fs');
var writeStream = fs.createWriteStream('./log.txt',
      {flags : 'a',
      encoding : 'utf8',
      mode : 0666});

writeStream.on('open', function() {
var counter = 0;

// obtém lista de arquivos
fs.readdir('./data/', function(err, files) {
```

```
// para cada arquivo
if (err) {
   console.error(err.message);
} else {
   files.forEach(function(name) {

     fs.stat('./data/' + name, function (err, stats) {

        if (err) return err;
        if (!stats.isFile()) {
           counter++;
           return;
        }

        // modifica o conteúdo
        fs.readFile('./data/' + name,'utf8', function(err,data) {

           if (err){
              console.error(err.message);
           } else {
              var adjData = data.replace(/somecompany/.com/g,
                      'burningbird.net');

              // grava em um arquivo
              fs.writeFile('./data/' + name, adjData,
                                             function(err) {

                 if (err) {
                    console.error(err.message);
                 } else {

                    // envia mensagem ao log
                    writeStream.write('changed ' + name + '\n'\n',
                     function(err) {

                       if(err) {
                         console.error(err.message);
                       } else {
                         console.log('finished ' + name);
                         counter++;
                         if (counter >= files.length) {
                            console.log('all done');
                         }
                       }
                    });
                 }
               });
              }
```

```
            });
          });
        });
      }
    });
});
writeStream.on('error', function(err) {
  console.error("ERROR:" + err);
});
```

Novamente, a aplicação atende o que se espera dela, e o faz muito bem – mas como é difícil de ler e manter! Usei `return` para alguns dos tratamentos de erro, eliminando um nível de aninhamento condicional, mas o programa ainda é quase impossível de manter. Já ouvi esse tipo de aninhamento de callbacks ser chamado de *callback spaghetti* (em português, "macarronada de callbacks") e até mesmo o colorido *pyramid of doom* (algo como "pirâmide do sofrimento eterno"), ambos termos bastante ilustrativos[1].

Os callbacks aninhados empurram o código para a margem direita do documento, tornando mais difícil assegurar que temos o código correto no devido callback. Entretanto, não é possível desmembrar o aninhamento de callbacks porque é essencial que os métodos sejam chamados apenas quando o anterior terminar:

1. Inicia a leitura do diretório.
2. Elimina da lista os subdiretórios.
3. Lê o conteúdo de cada arquivo.
4. Modifica o conteúdo.
5. Grava as modificações no arquivo original.

O que gostaríamos de fazer é encontrar uma maneira de implementar essa série de chamadas a métodos sem ter de depender de callbacks aninhados. Para isso, precisamos estudar módulos de terceiros e outras abordagens. No Capítulo 3, usaremos o módulo Async para demolir a pirâmide do sofrimento eterno. E, no Capítulo 9, veremos se as promessas do ES6 podem nos ajudar.

Outra maneira seria criar, para cada método, funções identificadas e as usar como funções

de callback no lugar das funções anônimas. Dessa forma, a pirâmide é bastante achatada e isso simplifica a depuração. Entretanto, fazer dessa forma não resolve alguns dos outros problemas, como determinar o momento em que cada processo termina. Para isso, ainda é necessário empregar um módulo que facilite o controle de funções assíncronas.

---

1 N. do T.: Em diversos livros, o leitor também encontrará o termo (bastante popular) *callback hell*, que significa "inferno de callbacks".

CAPÍTULO 3

Introdução aos módulos do Node e ao Node Package Manager (npm)

Ao longo de todo o livro teremos a oportunidade de trabalhar com inúmeros módulos do Node, a maioria deles *módulos nativos*. Esses módulos são chamados de nativos por serem instalados com o Node e podem ser incorporados a qualquer aplicação com o uso da instrução `require`.

Neste capítulo, exploraremos mais a fundo o conceito de módulos do Node, aprenderemos sobre a instrução `require` em detalhes e conheceremos o npm, o gerenciador de pacotes do Node. Também iremos às compras para obter módulos de terceiros quando precisarmos de funcionalidades que não estão nos módulos nativos. Muitas pessoas acreditam que alguns desses módulos de terceiros são essenciais para o desenvolvimento de aplicações em Node.

Introdução ao sistema de módulos do Node

A implementação básica do Node é mantida a mais enxuta possível. Em vez de incorporar cada componente diretamente no Node, os desenvolvedores deixam para implementar as funcionalidades adicionais em módulos à parte.

O sistema de módulos do Node é modelado a partir do *CommonJS module system*, uma maneira de criar módulos que garante a interoperabilidade total entre eles. O núcleo desse sistema é um acordo seguido à risca pelos desenvolvedores para assegurar que seus módulos trabalhem bem com os demais.

Dentre os requisitos do CommonJS module system implementados no Node, temos:

- Uma função `require` que recebe o identificador do módulo e devolve a API exportada.
- O nome do módulo é uma string e pode incluir caracteres barra (/) para a identificação de caminhos (paths).
- O módulo deve exportar explicitamente o que precisa ser exportado para fora do módulo.
- As variáveis do módulo são sempre privativas.

Algumas funcionalidades do Node são globais, o que significa que não é preciso fazer absolutamente nada para incluí-las. Entretanto, a maioria das funcionalidades do Node é incorporada usando o sistema de módulos.

Como o Node encontra um módulo e o carrega na memória

Quando queremos incluir acesso a um módulo do Node, seja um módulo nativo ou algum instalado por você fora da aplicação, use a instrução `require`:

```
var http = require('http');
```

Podemos acessar uma propriedade específica de um objeto exportado. Por exemplo, as pessoas frequentemente acessam somente a função `parse()` quando usam o módulo URL:

```
var name = require('url').parse(req.url, true).query.name;
```

Ou então podemos acessar um objeto específico do módulo para usar por toda a aplicação:

```
var spawn = require('child_process').spawn;
```

Quando sua aplicação requer um módulo, muitas coisas acontecem. Primeiro, o Node verifica se o módulo está *presente no cache*. Em vez de recarregar o módulo toda vez, o Node armazena em cache uma cópia do módulo na primeira vez em que é acessado, eliminando os atrasos decorrentes de ter de ler o módulo do disco novamente.

Correspondência um-para-um: o arquivo é o módulo

O Node permite apenas um módulo por arquivo.

Se o módulo chamado não estiver em cache, o Node verifica se é nativo. Os módulos nativos estão pré-compilados em formato binário, como os add-ons em C++ discutidos no Capítulo 1. Caso seja nativo, uma função é usada especificamente para ele e retorna a funcionalidade exportada.

Se o módulo não estiver em cache nem for nativo, um novo objeto Module é criado para ele e a propriedade `exports` do módulo é devolvida. Abordaremos os exports dos módulos em mais detalhes em "Criando e publicando seu próprio módulo do Node", mas estes basicamente retornam de forma pública a funcionalidade disponível para a aplicação.

O módulo, quando chamado, é colocado no cache. Se, por algum motivo, for necessário apagar o módulo do cache, pode-se, simplesmente:

```
delete require('./circle.js');
```

O módulo será recarregado na memória e no cache na próxima vez que a aplicação precisar dela.

Como parte do processo de chamar um módulo, o Node precisa encontrar onde o módulo está localizado. Ele executa uma sequência de verificações enquanto procura o arquivo que contém o módulo.

Primeiro, os módulos nativos têm prioridade. Por exemplo, podemos batizar um módulo nosso como `http`, mas, se tentarmos carregar o módulo `http`, o nosso será ignorado pelo Node, que carregará apenas a versão nativa. A única maneira de usar `http` como o nome do nosso módulo é informar também um caminho, para diferenciá-lo do módulo nativo:

```
var http = require ('/home/mylogin/public/modules/http.js');
```

O exemplo demonstra que, se informarmos um caminho absoluto (ou mesmo relativo) precedendo o nome de arquivo, o Node considera esse caminho. A declaração a seguir procura o módulo na mesma pasta em que a aplicação estiver (normalmente chamada de "pasta local" ou "diretório local"):

```
var someModule = require('./somemodule.js');
```

Costumo escrever a extensão do módulo, mas ela não é necessária. Quando informamos o nome sem a extensão, a primeira coisa que o Node faz é procurar o módulo na pasta local em um arquivo de extensão *.js*. Se existir, o módulo é carregado. Se não existir, outro arquivo, com extensão *.json*, é procurado na pasta local. Se um arquivo com o nome do módulo e a extensão *.json* existir, seu conteúdo é carregado como um objeto JSON. Por fim, o Node procura módulos com a extensão *.node*. Assume-se que esse módulo é um add-on pré-compilado do Node e este o trata adequadamente.

Os arquivos JSON não precisam de instruções *export* explícitas; basta que sejam arquivos JSON sintaticamente corretos.

Podemos usar um caminho relativo mais complexo:

```
var someModule = require('./somedir/someotherdir/somemodule.js');
```

Ou, ainda, um caminho absoluto, caso tenha certeza de que a aplicação jamais será transferida para outro local ou máquina. Este caminho é específico de um sistema de arquivos e não um URL:

```
var someModule = require('/home/myname/public/modules/somemodule.js');
```

Se o módulo for instalado pelo npm, não é necessário informar o caminho; basta escrever o nome do módulo:

```
var async = require('async');
```

O Node procura módulos em uma pasta chamada *node_modules*, usando uma sequência de leitura que inclui os seguintes locais:

1. Uma pasta *node_modules* dentro da pasta em que está a aplicação (*/home/meu_nome/projetos/node_modules*)
2. Uma pasta *node_modules* na pasta-mãe em relação à pasta atual da aplicação (*/home/meu_nome/node_modules*)
3. Subindo pela estrutura de diretórios, procurando *node_modules* em todas as pastas até atingir o nível mais alto (raiz) do sistema de arquivos (*/node_modules*)
4. Por fim, procura o módulo dentre os instalados globalmente (que discutiremos a seguir)

A razão pela qual o Node usa essa ordem de pesquisa é que as versões

localizadas de um módulo são acessadas antes das versões globais. Portanto, se estiver testando uma versão nova de um módulo e ele estiver instalado localmente em relação à aplicação:

```
npm install nome_do_módulo
```

este será carregado primeiro, em vez da versão global do mesmo módulo:

```
npm install -g nome_do_módulo
```

Podemos verificar quais módulos estão carregados usando a função `require.resolve()`:

```
console.log(require.resolve('async'));
```

A função retorna o local de instalação do módulo.

Caso o nome informado seja um nome de pasta e não de um arquivo, o Node procura um arquivo *package.json* contendo a propriedade `main` que indica o arquivo do módulo a ser carregado:

```
{ "name" : "nome_do_módulo",
"main" : "./lib/nome_do_módulo.js" }
```

Se o Node não encontrar um arquivo *package.json*, procurará arquivos *index.js* ou *index.node* e os carregará.

Caso todas as pesquisas falhem, será gerado um erro.

O cache reconhece nomes

Tenha em mente que as informações de cache são montadas com base no nome do arquivo e no caminho usado para carregar o módulo. Se uma versão global de um módulo já estiver no cache e uma versão local for carregada, ambas ficarão armazenadas no cache lado a lado.

Como o objeto Module é baseado em JavaScript, podemos olhar no código-fonte do Node para aprender o que acontece por trás dos panos.

Cada módulo envolvido com o objeto Module contém uma função `require`, de forma que, quando chamamos a função global `require`, ela imediatamente chama sua irmã gêmea específica dentro do módulo. Por sua vez, a função `Module.require()` chama outra função interna, `Module._load()`, que executa toda a funcionalidade que acabo de descrever. A única exceção é a solicitação para REPL, que veremos no próximo capítulo, e que tem sua própria maneira de fazer as coisas.

Se o módulo em questão for o principal, ou seja, o que é chamado na

linha de comando do sistema operacional (e que batizamos de "aplicação"), ele é na verdade associado a uma propriedade, `require.main`, do objeto global `require`. Digite a linha a seguir em um arquivo chamado *test.js* e execute-o com o Node:

```
console.log(require);
```

Será mostrado o objeto `main`, que é o objeto Module envolvendo o código da aplicação. Observe também que o nome do arquivo mostrado é exatamente o nome e caminho do arquivo que contém a aplicação. É mostrada também uma lista com os caminhos que o Node usou para carregar os módulos e o cache da aplicação, embora em nosso exemplo haja apenas uma linha, contendo a aplicação nesta instância.

Onde está o código-fonte do Node?

Podemos examinar o código-fonte do Node para entender sua funcionalidade; basta fazer seu download. Não precisa compilar esse código para instalar o Node em seu sistema, use-o apenas para aprendizado. A funcionalidade escrita em JavaScript está localizada na pasta */lib*, enquanto o código em C++ está em */src*.

Isso me leva a revisitar o conceito de objeto global do Node. No Capítulo 2, aprendemos sobre um objeto do Node chamado `global`, e vimos que é diferente do objeto global usado no navegador. Ao contrário do navegador, as variáveis de nível mais alto estão, no Node, restritas a seu contexto imediato, o que significa que as variáveis declaradas em um módulo não conflitarão com as declaradas na aplicação ou em qualquer outro módulo incluído na aplicação. Isso acontece porque o Node envolve, na função a seguir, todos os scripts:

```
function (module, exports, __filename, ...) {}
```

Em outras palavras, o Node embrulha os módulos (seja a aplicação, seja qualquer outro) em funções anônimas, expondo apenas o que o desenvolvedor do módulo quis realmente expor. Como essas propriedades dos módulos são precedidas pelo nome do módulo quando este é usado, não há como ocorrer conflitos com as variáveis declaradas localmente.

Falaremos mais sobre contexto a seguir.

O módulo VM e as "caixas de areia"

Uma das primeiras coisas que aprendemos sobre JavaScript é que devemos evitar o uso de `eval()` a qualquer custo. A razão para tal é que `eval()` executa o JavaScript no mesmo contexto que o resto da aplicação. Em algumas situações, temos de lidar com código arbitrário ou desconhecido[1], e o uso de `eval()` fará com que esse código não confiável seja executado no mesmo nível de confiança que o código cuidadosamente escrito da aplicação. Comparando, seria o mesmo que pegar um texto digitado pelo usuário em um campo input do HTML e entregá-lo diretamente a uma query SQL sem a devida verificação de segurança.

Se, por alguma razão, for necessário executar um bloco externo de código em JavaScript, vindo de alguma fonte externa e não confiável, podemos fazê-lo usando o módulo VM como uma sandbox (caixa de areia). Contudo, os próprios desenvolvedores do Node sugerem que essa técnica não é confiável. A única maneira realmente segura de executar código arbitrário em JavaScript é empregando um processo separado. Se a origem do código externo em JavaScript for considerada confiável, ainda assim é interessante isolar de seu ambiente local o script desconhecido, evitando acidentes e outros dissabores.

Os scripts podem ser pré-compilados, usando o objeto `vm.Script`, ou repassados como parte de uma função chamada diretamente por `vm`. Existem três tipos de função. O primeiro tipo, que pode ser `script.runInNewContext()` ou `vm.runInNewContext()`, executa o script no novo contexto sem que tenha acesso às variáveis locais ou ao objeto global. Em vez disso, é criado uma sandbox *contextualizado*, que é passado à função. O código a seguir demonstra esse conceito. A sandbox contém dois valores globais – dois nomes, que são idênticos aos nomes dos dois objetos globais do Node, só que redefinidos:

```
var vm = require('vm');

var sandbox = {
   process: 'this baby',
   require: 'that'
};

vm.runInNewContext('console.log(process);console.log(require)',sandbox);
```

O código anterior causa um erro, porque o objeto `console` não é parte do contexto de execução do script. Para que isso ocorra, o código deve ficar assim:

```
var vm = require('vm');

var sandbox = {
   process: 'this baby',
   require: 'that',
   console: console
};

vm.runInNewContext('console.log(process);console.log(require)',sandbox);
```

O problema é que fazer isso destrói completamente o que queríamos desde o começo: criar um novo contexto para o script. Se for realmente necessário que o script tenha acesso ao console (ou qualquer outro objeto global), use `runInThisContext()`. No Exemplo 3.1, o objeto Script é usado para demonstrar como o contexto inclui o objeto global, mas não os objetos locais.

Exemplo 3.1 – Executando um script que acessa o console global

```
var vm = require('vm');

global.count1 = 100;
var count2 = 100;

var txt = 'if (count1 === undefined) var count1 = 0; count1++;' +
   'if (count2 === undefined) var count2 = 0; count2++;' +
   'console.log(count1); console.log(count2);';

var script = new vm.Script(txt);
script.runInThisContext({filename: 'count.vm'});

console.log(count1);
console.log(count2);
```

Ao rodar a aplicação, o resultado é:

```
101
1
101
100
```

A variável `count1` é declarada no objeto global e fica também disponível no contexto de execução do script. A variável `count2` é local e deve ser definida

dentro do contexto. Qualquer mudança na variável local, que pertence ao script dentro do sandbox, não causa impacto na variável local de mesmo nome presente na aplicação.

Um erro será gerado se não declararmos `count2` no script em execução no contexto separado. O erro é mostrado porque uma das opções da função dentro do sandbox é `displayErrors`, que por default tem o valor `true`. As outras opções de `runInThisContext()` são `filename`, mostrada no exemplo, e `timeout`, o tempo em milissegundos no qual o script tem permissão para executar antes de ser encerrado (e o encerramento levanta um erro). A opção `filename` foi usada para especificar o nome do arquivo que aparece no rastreamento de pilha (stack trace) quando o script está em execução. Entretanto, se for necessário especificar um nome de arquivo para o objeto Script, é necessário informá-lo quando esse objeto for criado, não no contexto em que a função é chamada:

```
var vm = require('vm');

global.count1 = 100;
var count2 = 100;

var txt = 'count1++;' +
          'count2++;' +
          'console.log(count1); console.log(count2);';

var script = new vm.Script(txt, {filename: 'count.vm'});

try {
   script.runInThisContext();
} catch(err) {
   console.log(err.stack);
}
```

Tirando a diferença em `filename`, o módulo Script suporta as outras duas opções globais dentro dos contextos de chamadas a funções: `displayErrors` e `timeout`.

Executar o código resulta em um erro porque o script dentro do sandbox não tem acesso à variável local (`count2`) da aplicação, embora tenha acesso à variável global `count1`. Quando esse erro é mostrado, o *stack trace* aparece logo em seguida, mostrando o nome do arquivo passado como uma opção.

Em vez de escrever o código arbitrário diretamente na aplicação, podemos lê-lo de um arquivo. Suponha que queiramos executar em nossa aplicação o seguinte script externo, gravado em um arquivo chamado *script.js*:

```
if (count1 === undefined) var count1 = 0; count1++;
if (count2 === undefined) var count2 = 0; count2++;
console.log(count1); console.log(count2);
```

Para pré-compilar e rodar o script em um sandbox, considere o seguinte código como sendo a aplicação:

```
var vm = require('vm');
var fs = require('fs');

global.count1 = 100;
var count2 = 100;

var script = new vm.Script(fs.readFileSync('script.js','utf8'));
script.runInThisContext({filename: 'count.vm'});

console.log(count1);
console.log(count2);
```

Algo nos impede de usar o módulo File System diretamente no script externo. A resposta é: o script está atribuído a uma variável local e não acessível. E por que não podemos simplesmente importar o módulo File System diretamente no script? Porque, nesse objeto, a instrução `require` não está disponível. Nenhum dos objetos ou funções globais, incluindo o `require`, está acessível para o script.

A última função de sandbox é `runInContext()`. Ela também recebe um sandbox, mas este precisa, necessariamente, ser *contextualizado* (ou seja, o contexto precisa ser explicitamente criado) antes da chamada à função. No exemplo a seguir, ele será chamado diretamente pelo objeto VM. Observe que uma variável adicional foi incluída no código dentro do sandbox contextualizado, que em seguida a mostra na aplicação:

```
var vm = require('vm');
var util = require('util');

var sandbox = {
   count1 : 1
};

vm.createContext(sandbox);
if (vm.isContext(sandbox)) console.log('contextualized');
```

```
vm.runInContext('count1++; counter=true;',sandbox,
   {filename: 'context.vm'});

console.log(util.inspect(sandbox));
```

O resultado dessa aplicação é:

```
contextualized
{ count1: 2, counter: true }
```

A função `runInContext()` suporta as três opções também usadas por `runInThisContext()` e `runInNewContext()`. Novamente, a diferença entre executar as funções via módulo Script ou diretamente pelo módulo VM é que o objeto `Script` pré-compila o código e recebe o nome de arquivo quando o objeto é criado, em vez de estar em uma das opções da chamada à função.

Executando o script em um processo à parte

Se for interessante ampliar o isolamento do script arbitrário, podemos executá-lo em um processo adicional, separado da aplicação principal. Para isso, pesquise sobre os módulos de sandbox de terceiros que ofereçam essa proteção adicional. A próxima seção descreve como encontrá-los.

Conhecendo o NPM a fundo

Muito da rica funcionalidade associada ao Node está relacionado ao uso de módulos de terceiros. Há módulos de roteamento, módulos para trabalhar com bancos de dados relacionais ou baseados em documentos, módulos para templates, módulos de teste e até mesmo módulos para serviços de pagamento online.

GitHub

Embora não seja obrigatório, todos os desenvolvedores são encorajados a disponibilizar seus módulos no GitHub.

Para usar um módulo, podemos baixar o código-fonte e instalá-lo manualmente no ambiente da aplicação. Muitos módulos oferecem instruções básicas de instalação ou, no mínimo, podemos deduzir os requisitos de instalação examinando os arquivos e pastas que compõem o módulo. Todavia, há uma maneira muito mais fácil de instalar um módulo do Node: usando npm.

O site do npm é *http://npmjs.org/*. A documentação está disponível em uma página no mesmo site.

O Node já vem com o npm instalado, mas nem sempre na versão mais recente. Se for necessário atualizá-lo, use o comando a seguir (com `sudo`, se for o apropriado para seu ambiente):

```
npm install npm -g
```

Cuidado ao atualizar o npm: todos os membros da equipe precisam usar a mesma versão, caso contrário pode haver resultados inesperados.

Para uma visão geral sobre os comandos do npm, use:

```
$ npm help npm
```

Os módulos podem ser instalados global ou localmente. A instalação local é a melhor opção caso estejamos trabalhando em um projeto isolado e os outros membros da equipe não necessitem desse módulo. Uma instalação local, que é o comportamento default, instala o módulo na pasta atual, dentro da subpasta *node_modules*.

```
$ npm install nome_do_módulo
```

Como exemplo, para instalar o módulo Request, o comando é:

```
$ npm install request
```

O npm não apenas instala o Request, mas também descobre as dependências desse módulo, e as dependências das dependências, e instala todas. Quanto mais sofisticado o módulo, mais dependências de outros módulos têm de ser instaladas. A Figura 3.1 mostra uma lista parcial gerada pela instalação do módulo Request em minha máquina Windows (a versão do Node neste exemplo é 5.0.0).
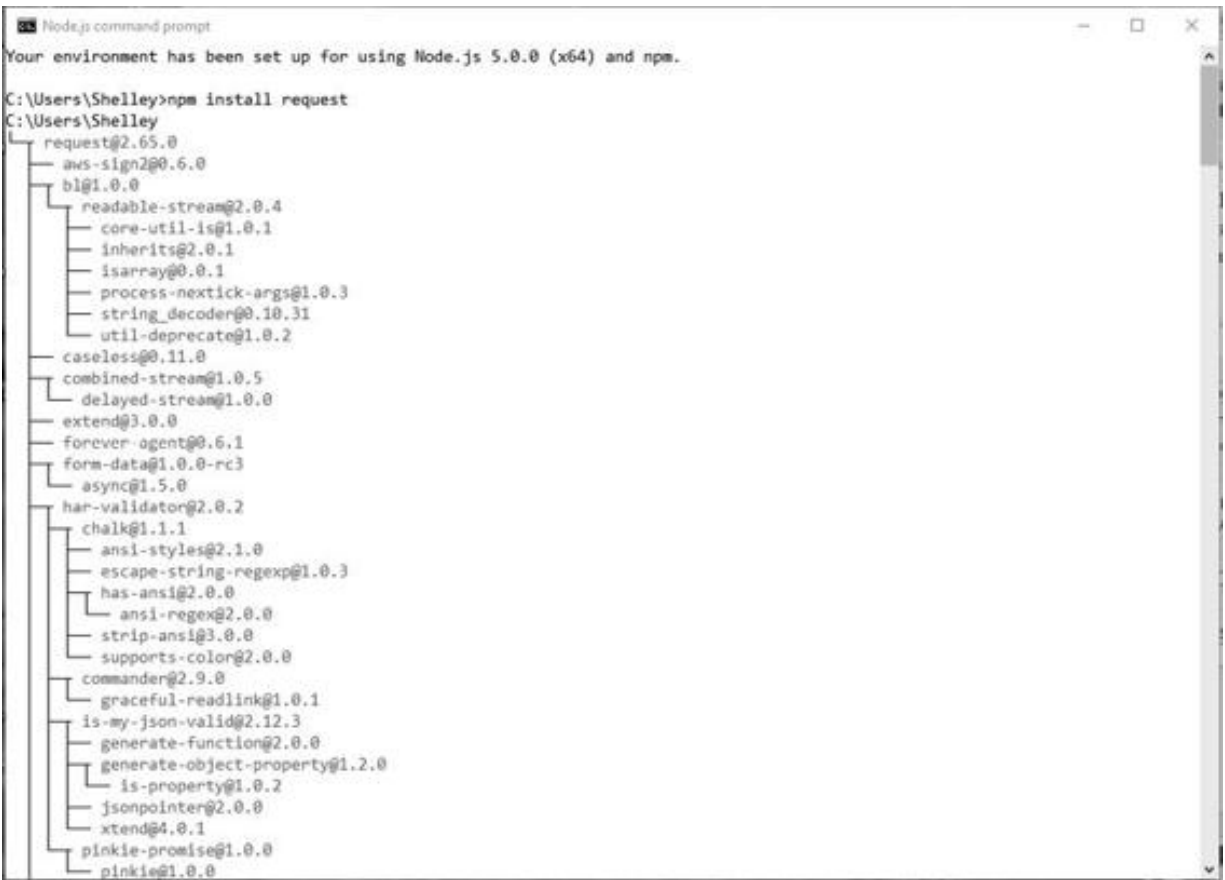
Figura 3.1 – Lista parcial das dependências instaladas junto com o módulo Request.

Uma vez instalado o módulo, podemos encontrá-lo na pasta local *node_modules/request*. Quaisquer dependências de Request estarão instaladas em uma nova pasta *node_modules/request/node_modules*. Uma rápida olhada mostra que ganhamos muita funcionalidade só com a instalação de um módulo como o Request.

Para instalar o pacote globalmente, use a opção `-g` ou `--global`:

```
$ npm install request -g
```

No Linux, não se esqueça de usar `sudo` para instalar o módulo globalmente:

```
$ sudo npm install request -g
```

Alguns módulos, incluindo aplicações de linha de comando, obrigatoriamente devem ser instalados de forma global. Esses exemplos instalam pacotes registrados no site do npm. Podemos também instalar

um pacote ou módulo que está numa pasta do sistema de arquivos, ou um arquivo *.tar* que pode estar tanto local quanto transferido de um URL:

```
npm install http://nome_da_empresa.com/nome_do_módulo.tgz
```

Se o pacote disponibilizar várias versões, podemos instalar uma versão específica:

```
npm install módulo@0.1
```

Dá até para instalar nosso velho amigo jQuery:

```
npm install jquery
```

Se não estiver mais usando o módulo, podemos desinstalá-lo:

```
npm uninstall nome_do_módulo
```

O comando a seguir diz ao npm para verificar a existência de versões mais novas dos módulos instalados e atualizá-los se existirem:

```
npm update
```

Podemos atualizar um único módulo:

```
npm update nome_do_módulo
```

e até mesmo o próprio npm:

```
npm install npm -g
```

Para verificar se algum pacote está desatualizado, o comando é:

```
npm outdated
```

O comando pode, como qualquer outro, ser usado em um único módulo.

Para listar os pacotes instalados e suas dependências, use `list`, `ls`, `la` ou `ll`:

```
npm ls
```

As opções `la` e `ll` mostram descrições detalhadas. Por exemplo, uma das dependências de Request é tunnel-agent@0.4.1 (versão 0.4.1 do pacote tunnel-agent). Mas que diabos é tunnel-agent? Se rodarmos `npm la request` na linha de comando, obteremos uma lista com todas as dependências, incluindo tunnel-agent, e detalhes de cada um:

```
tunnel-agent@0.4.1
HTTP proxy tunneling agent. Formerly part of mikeal/request,
now a standalone module
```

```
git+https://github.com/mikeal/tunnel-agent.git
https://github.com/mikeal/tunnel-agent#readme
```

Às vezes a saída mostrará um aviso, como, por exemplo, uma dependência não satisfeita ou a necessidade de um módulo mais antigo. Para corrigir, instale o módulo faltante ou a versão correta do módulo existente:

```
npm install jsdom@0.2.0
```

Podemos instalar diretamente todas as dependências com a chave `-d`. Por exemplo, na pasta do módulo, digite:

```
npm install -d
```

Se for necessário instalar uma versão de módulo que ainda não foi cadastrada no repositório central do npm, é possível instalá-la diretamente de seu repositório Git:

```
npm install https://github.com/visionmedia/express/tarball/master
```

Tenha cuidado ao instalar uma versão ainda não lançada de um módulo. Se executarmos um npm update, essa versão pode ser substituída pela versão anterior que está registrada no repositório central do npm.

Usando o npm no Linux via PuTTY

Se sua estação de trabalho for Windows, mas estiver trabalhando com a dupla Node/npm em uma máquina Linux distante (seja ela remota ou virtual) usando o emulador de terminal PuTTY, não conseguimos listagens e mensagens de saída legíveis dos comandos do npm. Em vez de linhas bem definidas mostrando as dependências, a tela mostra caracteres estranhos.

Para resolver o problema, é preciso configurar o PuTTY para traduzir os caracteres usando UTF-8. No PuTTY, clique na opção **Window->Translation** e selecione **UTF-8** a partir do menu drop-down.

Para ver quais módulos estão instalados globalmente, use:

```
npm ls -g
```

Podemos aprender mais sobre a instalação do npm usando o comando `config`. O exemplo a seguir lista as configurações do npm:

```
npm config list
```

Podemos obter uma visão mais ampla das configurações do npm com:

```
npm config ls -l
```

Podemos modificar ou remover itens de configuração com comandos diretos:

```
npm config delete keyname
npm config set keyname value
```

ou editando diretamente o arquivo de configuração:

```
$ npm config edit
```

Recomendo-lhe com veemência que deixe a configuração no npm como está, a não ser que tenha absoluta certeza de que sabe o que está fazendo.

Podemos ainda procurar um módulo usando quaisquer termos de busca:

```
npm search html5 parser
```

Na primeira busca, o npm cria um índice, o que pode levar alguns minutos. Quando terminar, temos uma lista com os possíveis módulos que combinam com os termos informados.

Sobre o erro "registry error parsing json"

Se o npm reclamar de "registry error parsing json" (erro no registro ao analisar dados JSON), podemos usar um mirror alternativo do npm para completar a tarefa. Por exemplo, para usar um mirror europeu, o comando é:

```
npm --registry http://registry.npmjs.eu/ search html5 parser
```

O site do npm mantém um cadastro (em inglês, registry) dos módulos disponíveis e uma lista com os mais usados – ou seja, os módulos mais chamados por outros módulos, em dependências, ou por aplicações do Node. Mais adiante, veremos alguns deles.

Uma última observação sobre o npm antes de seguirmos adiante. Quando começamos a brincar com o npm, percebemos algumas mensagens de alerta no final da listagem de saída do comando. A primeira linha avisa que não conseguiu encontrar um arquivo *package.json* e as demais informam erros diversos associados à ausência desse *package.json*.

A documentação do npm recomenda que criemos um arquivo *package.json* para manter as dependências locais. Não é um requisito irritante, ao contrário das constantes mensagens de erro.

Para criar um arquivo *package.json* default na pasta do projeto, execute o comando:

```
npm init --yes
```

O arquivo *package.json* é criado na pasta atual, e o npm faz algumas

perguntas básicas sobre o projeto, como, por exemplo, o nome, e cada pergunta tem um valor default. A partir desse ponto, quando um módulo for instalado, não seremos mais agraciados com as mensagens irritantes. Entretanto, para eliminar algumas delas será necessário atualizar os dados em formato JSON presentes no arquivo para incluir uma descrição (chave `description`) e um repositório (chave `repository`). Também pode ser necessário atualizar o arquivo para incluir um módulo instalado recentemente, e para tal usamos a sintaxe a seguir:

```
npm install request --save-dev
```

Com isso, o nome e a versão do módulo são gravados no campo `devDependencies` do arquivo *package.json*. Podemos salvar o módulo também para as dependências de produção, mas veremos isso com mais detalhes quando esmiuçarmos o arquivo *package.json* (em "Criando e publicando seu próprio módulo do Node").

Para salvar automaticamente as dependências, é preciso editar o arquivo *npmrc* (ou criar um, se ele não existir). É possível criá-lo por usuário (*~/.npmrc*), por projeto (*/caminho/projeto/.npmrc*), globalmente (*$PREFIX/etc/npmrc*) e até mesmo usando o arquivo de configuração principal do npm (*/caminho/para/o/npm/npmrc*). O comando a seguir edita as configurações do usuário para salvar as dependências automaticamente:

```
npm config set save=true
npm config set save-exact=true
```

O que o comando faz é, automaticamente, adicionar a opção `--save` (para gravar o pacote nas dependências) e uma opção `--save-exact` (gravar com a versão exata, não o padrão de semver do npm) ao instalar novos pacotes.

Há também muitas configurações diferentes que podem ser ajustadas. Um belo lugar para conhecer melhor todas elas é na documentação do npm.

Criando e publicando seu próprio módulo do Node

Assim como nos programas em JavaScript que rodam no navegador, é útil

retirar do programa principal quaisquer partes do código que possam ser reutilizadas e criar, com elas, bibliotecas que possam ser utilizadas várias vezes na mesma aplicação ou em outras. A única diferença é que, no Node, existem alguns passos a mais, pois é necessário converter a biblioteca JavaScript criada por nós em um módulo.

Criando um módulo

Digamos que em uma biblioteca JavaScript exista a função `concatArray`, que recebe uma string e um array de strings. A função concatena a primeira string com cada elemento da sequência, devolvendo um novo array:

```
function concatArray(str, array) {
   return array.map(function(element) {
      return str + ' ' + element;
   });
}
```

Então, imagine que queiramos usar essa função, bem como outras na mesma biblioteca, em nossas aplicações em Node.

Para converter a biblioteca JavaScript de forma que o Node possa usá-la, é preciso exportar todas as funções que devem ser expostas; para isso usamos o objeto `exports`, como mostrado no código a seguir:

```
exports.concatArray = function(str, array) {
   return array.map(function(element) {
      return str + ' ' + element;
   });
};
```

Para usar a função `concatArray` em uma aplicação escrita em Node, basta importar a biblioteca. A função exportada pode, agora, ser chamada:

```
var newArray = require ('./arrayfunctions.js');

console.log(newArray.concatArray('hello', ['test1','test2']));
```

Podemos até criar um módulo consistindo de um construtor ou função e exportá-lo usando `module.exports`.

Por exemplo, o módulo Mime, do qual muitos outros módulos dependem, cria uma função `Mime()`:

```
function Mime() { ... }
```

e adiciona funcionalidade a ela usando a propriedade `prototype`:

```
Mime.prototype.define = function(map) {...}
```

para depois criar uma instância default:

```
var mime = new Mime();
```

atribuir a função `Mime` a sua própria propriedade de mesmo nome:

```
mime.Mime = Mime;
```

e, por fim, exportar a instância:

```
module.exports=mime;
```

Depois disso, podemos usar as inúmeras funções de mime em qualquer aplicação:

```
var mime = require('mime');
console.log(mime.lookup('phoenix5a.png')); // image/png
```

Empacotando uma pasta por completo

Podemos dividir o código do módulo em diversos arquivos JavaScript (em vez de usar um arquivão só), todos localizados dentro de uma mesma pasta. O Node é capaz de carregar na memória o conteúdo de toda a pasta, desde que ela esteja organizada da forma correta. Há duas formas de organização possíveis:

A primeira é criar um arquivo *package.json* com informações sobre a pasta. A estrutura de dados contém outras informações, mas os dois itens relevantes no tocante ao empacotamento são `name` e `main`:

```
{ "name" : "mylibrary",
   "main" : "./mymodule/mylibrary.js"
}
```

A primeira propriedade, `name`, é o nome do módulo. A segunda, `main`, indica o ponto de entrada do módulo.

A segunda maneira de carregar todo o conteúdo da pasta é incluir um arquivo, que pode ter o nome de *index.js* ou *index.node*, na pasta, e esse arquivo funcionará como o ponto de entrada do módulo.

E por que desejaríamos dividir nosso módulo em vários arquivos numa

pasta em vez de um único arquivo que contenha tudo? A razão mais frequente é quando o módulo usa bibliotecas JavaScript preexistentes, e nosso código é apenas um arquivo "wrapper" que envolve as funções expostas com instruções `exports`. Outra razão para fazer isso é quando o código da biblioteca é tão grande e complexo que é muito mais simples modificar ou fazer manutenção com tudo dividido em arquivos pequenos.

Independentemente do porquê, tenha em mente que todos os objetos exportados devem estar reunidos em um único arquivo principal, o ponto de entrada, pois o que o Node realmente chama é esse ponto de entrada.

Preparando o módulo para publicação

Se for interessante disponibilizar o pacote para que outros programadores usem, podemos simplesmente hospedá-lo e promovê-lo em um website, mas dessa forma deixaremos de informar uma parcela significativa do público pretendido. Quando um módulo está maduro o suficiente para ser disponibilizado ao grande público, a decisão mais acertada é cadastrá-lo no repositório do npm, o chamado npm registry.

O arquivo *package.json*, mencionado anteriormente, é baseado na CommonJS, um conjunto de recomendações para módulos e sistemas em JavaScript. A documentação do JSON usado pelo npm pode ser encontrada online.

Dentre os campos que se recomenda incluir no arquivo *package.json* estão:

`name`
: O nome do pacote – obrigatório.

`description`
: A descrição do pacote.

`version`
: A versão atual do pacote, de acordo com as exigências semânticas de versionamento – obrigatório.

`keywords`
: Um array contendo termos relacionados ao módulo, para fins de busca.

`maintainers`
: Um array contendo os dados de contato dos mantenedores do pacote (inclui nome, email e site).

`contributors`
: Um array contendo os dados de contato dos desenvolvedores que contribuíram no pacote (inclui nome, email e site).

`bugs`
: O URL no qual as falhas encontradas devem ser reportadas.

`licenses`
: Um array contendo as licenças de uso pertinentes.

`repository`
: O repositório do pacote.

`dependencies`
: Pacotes que são pré-requisito para o funcionamento deste, com seus números de versão.

Apenas os campos `name` e `version` são obrigatórios, embora recomenda-se que todos os campos sejam preenchidos. Felizmente, o npm facilita a criação desse arquivo. O comando a seguir auxilia na tarefa:

```
npm init
```

A ferramenta varrerá todos os campos, obrigatórios ou não, perguntando o valor para cada um deles. Quando o questionário termina, o arquivo *package.json* é gerado.

No Capítulo 2, mais especificamente no Exemplo 2.7, criamos um objeto chamado `InputChecker` que analisa os dados entrantes e, caso identifique algum comando, processa-o. O exemplo demonstra como incorporar um `EventEmitter`. Vamos modificar esse objeto simples para torná-lo reutilizável por quaisquer outras aplicações ou módulos.

Primeiro, criaremos uma pasta *inputcheck* dentro de *node_modules* e transferiremos o arquivo `InputChecker`, que contém o código desse objeto, para lá. O arquivo precisa ser renomeado como *index.js*. Depois, precisamos modificar o código para retirar o trecho que implementa o novo objeto. Grave esse trecho em um arquivo separado, pois será usado para testes futuramente. A última modificação é adicionar um objeto `exports`, o que resulta no código mostrado no Exemplo 3.2.

Exemplo 3.2 – Aplicação do Exemplo 2.7 modificada para se tornar um módulo

```
var util = require('util');
var eventEmitter = require('events').EventEmitter;
var fs = require('fs');

exports.InputChecker = InputChecker;

function InputChecker(name, file) {
   this.name = name;
   this.writeStream = fs.createWriteStream('./' + file + '.txt',
      {'flags' : 'a',
       'encoding' : 'utf8',
       'mode' : 0666});
};

util.inherits(InputChecker,eventEmitter);
InputChecker.prototype.check = function (input) {
   var command = input.toString().trim().substr(0,3);
   if (command == 'wr:') {
      this.emit('write',input.substr(3,input.length));
   } else if (command == 'en:') {
      this.emit('end');
   } else {
      this.emit('echo',input);
   }
};
```

Não podemos exportar a função do objeto diretamente, porque `util.inherits` espera que o objeto exista dentro do arquivo `InputChecker`. É preciso ainda modificar o protótipo `prototype` do objeto `InputChecker` mais adiante no arquivo. Poderíamos ter simplesmente alterado as referências no código para usar `exports.InputChecker`, mas isso é *gambiarra*. É tão fácil quanto, e muito mais elegante, atribuir o objeto em uma instrução à

parte.

Para criar o arquivo *package.json*, usamos `npm init` para responder a todas as perguntas. O arquivo resultante está no Exemplo 3.3.

Exemplo 3.3 – Arquivo package.json gerado de forma assistida para o módulo inputChecker

```
{
   "name": "inputcheck",
   "version": "1.0.0",
   "description": "Looks for and implements commands from input",
   "main": "index.js",
   "scripts": {
      "test": "echo \"Error: no test specified\" && exit 1"
   },
   "keywords": [
      "command",
      "check"
   ],
   "author": "Shelley Powers",
   "license": "ISC"
}
```

O comando `npm init` não solicita as dependências, portanto teríamos de adicioná-las manualmente ao arquivo. Uma maneira mais elegante seria usar `npm install --save` no momento da instalação dos módulos, o que automaticamente os marca como dependências do nosso módulo. Entretanto, o módulo `InputChecker` não tem nenhuma dependência externa, portanto podemos deixar esses campos em branco.

Neste ponto, podemos testar o novo módulo para nos certificar de que realmente funciona como um módulo. No Exemplo 3.4 temos aquele trecho, extraído do `InputChecker` original, que testava o novo objeto. Agora, esse trecho de código reside em um arquivo separado.

Exemplo 3.4 – Aplicação de teste para InputChecker

```
var inputChecker = require('inputcheck').InputChecker;

// testa o novo objeto e o tratamento de eventos
var ic = new inputChecker('Shelley','output');

ic.on('write', function(data) {
```

```
    this.writeStream.write(data, 'utf8');
});

ic.addListener('echo', function( data) {
    console.log(this.name + ' wrote ' + data);
});

ic.on('end', function() {
    process.exit();
});

process.stdin.resume();
process.stdin.setEncoding('utf8');
process.stdin.on('data', function(input) {
   ic.check(input);
});
```

Podemos agora transferir a aplicação de teste para uma nova pasta chamada *test*, dentro da pasta principal de nosso módulo. Essa aplicação será empacotada com nosso módulo como exemplo. A boa prática manda que sempre exista em nosso módulo uma pasta *test* com uma ou mais aplicações de teste e uma pasta *doc* contendo a documentação. Como nosso módulo é muito pequeno, um arquivo *README* é mais que suficiente.

Como agora temos uma aplicação de teste, é preciso incluir uma referência a ela no arquivo *package.json*:

```
"scripts": {
    "test": "node test/test.js"
},
```

O teste proposto aqui é bastante primitivo e não faz uso de um dos ambientes de teste do Node, que são bastante robustos. Contudo, já serve para demonstrar como um teste deve ser. Para executar a aplicação de teste, digite o seguinte comando no terminal:

```
npm test
```

O último passo é criar um arquivo *.tar.gz* empacotando todo o módulo. Se preferir, pode pular esse passo quando publicar seu módulo para o público em geral, que discutiremos em “Publicando seu módulo”.

Tarball? Bola de piche? O que é isso?

Na literatura em língua inglesa, é muito comum encontrar o termo *tarball*, que em

português poderia ser traduzido como "bola de alcatrão" ou "bola de piche". Quem nunca trabalhou com Unix pode achar esse termo bastante estranho, especialmente em um livro que não versa sobre asfalto. Um tarball nada mais é do que um ou mais arquivos e pastas empacotados juntos em um único arquivo usando o comando `tar`.

Agora que temos todo o necessário, podemos publicar o módulo.

Oferecendo mais que um mero módulo

Para nosso próprio uso, podemos passar muito bem com um módulo simplista, como é o caso do nosso `inputcheck`. Porém, se o objetivo for disponibilizá-lo na internet para todos, precisamos oferecer muito mais. É necessário um repositório de onde as pessoas possam baixá-lo, um URL para que as falhas possam ser reportadas, um site oficial para o módulo e outros itens semelhantes. Ainda assim, a maneira mais fácil de conseguir isso é começar simples e ir construindo a partir dessa base.

Publicando seu módulo

O pessoal que nos agraciou com o npm também providenciou um pequeno manual detalhando o que é necessário para publicar um módulo para o Node: o Guia do Desenvolvedor, mais facilmente encontrável pelo seu nome em inglês, Developer Guide.

A documentação especifica alguns requisitos adicionais para o arquivo *package.json*. Além dos campos já criados, será necessário incluir um campo `directories` com um hash das pastas existentes no módulo, como, por exemplo, as já mencionadas *test* e *doc*:

```
"directories" : {
   "doc" : ".",
   "test" : "test",
   "example" : "examples"
}
```

Antes de publicar, o Guia recomenda que testemos se o módulo consegue ser instalado de forma limpa. Para testar, digite o comando a seguir na pasta-raiz do módulo:

```
npm install . -g
```

Neste ponto, o módulo `inputChecker` foi testado, o arquivo *package.json* modificado e confirmamos que o pacote é instalado com sucesso.

Em seguida, é necessário adicionar a nós mesmos como usuário do npm, se ainda não o tivermos feito. O comando para tal é:

```
npm adduser
```

O npm mostra um questionário perguntando o nome do usuário, uma senha e um endereço de email válido.

Há uma última coisa que podemos fazer:

```
npm publish
```

É possível fornecer o caminho para o arquivo *.tar.gz* ou para a pasta-raiz do módulo. Tenha em mente que, como está expressamente alertado no Guia, todo o conteúdo da pasta-raiz do módulo e suas subpastas é exposto ao mundo externo, a não ser que tenhamos providenciado um arquivo `.npmignore` que lista os arquivos que devem ser ignorados. Os arquivos listados em `.npmignore` devem ter sido registrados em *package.json*. Em vez disso, o melhor é remover tudo o que não for necessário antes de publicar o módulo.

Uma vez publicado – e uma vez que o código-fonte tenha sido hospedado no repositório que você estiver usando (por exemplo, o GitHub) –, o módulo está, oficialmente, disponível para qualquer programador do planeta. Faça publicidade de seu módulo no Twitter, Google+, Facebook, no seu site pessoal e onde quer que você acredite que existam pessoas interessadas em conhecê-lo. Esse tipo de publicidade não é considerado uma demonstração de ego inflado nem uma variação do pecado da soberba – pelo contrário, será recebido como um ato de *altruísmo*, de doação para a coletividade.

Descobrindo módulos interessantes para o Node, incluindo três imperdíveis

Não é nada difícil encontrar módulos para o Node. Muitos deles serão recomendados por amigos ou colegas de trabalho. Outros aparecerão quando procurarmos na internet uma funcionalidade específica. Nesse caso, normalmente os mais populares aparecem no topo da lista dos mecanismos de busca.

Houve uma época em que o número de módulos era tão pequeno que eram manualmente listados por tipo em uma única página do site oficial

do npm. Esses dias pioneiros são, hoje, apenas lembrança. A maneira moderna de procurar módulos é usando o mecanismo de busca do próprio site do npm.

Por exemplo, se quisermos procurar módulos que suportem a autenticação pelo protocolo OAuth, basta usar esse termo na busca, no topo da página. O primeiro resultado é, simplesmente, OAuth, que é o nome do módulo mais popular para a tarefa hoje. O nível de popularidade pode ser atestado pelas estatísticas, mostradas na descrição de cada módulo.

Sempre verifique as estatísticas dos módulos. O maior desafio para escolher qual módulo usar é que muitos deles não são mais suportados ou sua qualidade é questionável. A única maneira de saber se o módulo "presta", ou seja, se está sendo ativamente suportado e usado por um grande número de desenvolvedores, é comparando as estatísticas listadas à direita do nome do módulo. A Figura 3.2 mostra um exemplo, justamente o módulo OAuth.
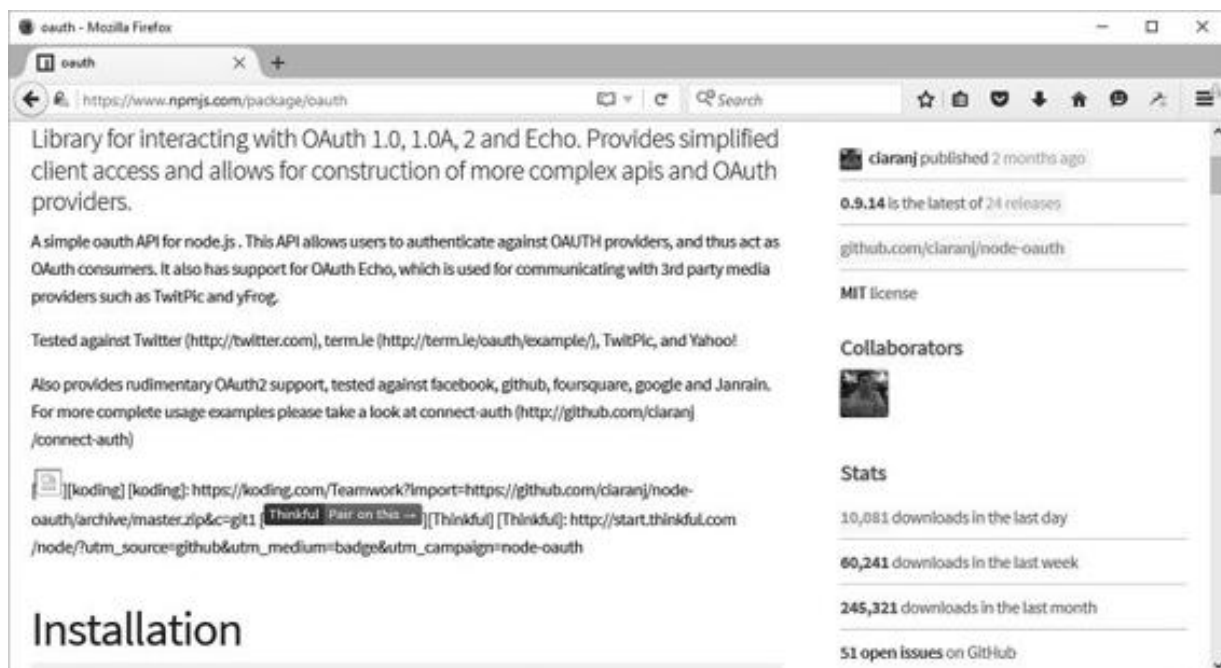
Figura 3.2 – Estatísticas do NPM para o módulo OAuth.

Observe o texto realçado. O módulo disponível hoje foi publicado pela última vez há menos de 60 dias, portanto demonstra ser ativamente

atualizado. É a mais recente dentre um conjunto de 24 versões, o que mostra grande atividade por parte da equipe de desenvolvimento. Contudo, o item que decisivo é o número de downloads: mais de dez mil em um único dia. Portanto, o módulo é usado ativamente por milhares de desenvolvedores, o que indica sua viabilidade e que não deixará de ser mantido tão cedo.

Além disso, podemos visitar a página do GitHub para o módulo, verificar a página de "issues" (problemas reportados) e ter uma ideia do quão comprometidos estão os desenvolvedores do módulo em resolver os problemas relatados pelos programadores que usam o módulo. Também devemos levar em conta a severidade e o tipo dos problemas relatados. Dependendo do problema, mesmo um módulo popular pode não ser uma boa escolha (por exemplo, uma falha grave de segurança).

Não há nenhuma dúvida de que o módulo OAuth é o melhor hoje, sendo ativamente desenvolvido dentre todos os cadastrados no npm, tanto pelo suporte ativo quanto pela excelência de código. É claro que existem outras opções também muito boas, mas, enquanto não estiver bem familiarizado com o Node, o melhor a fazer é usar os módulos mais populares e festejados.

Aliás, o site do npm lista em sua página inicial os módulos mais populares do dia. Há ainda outra lista com os módulos mais bem cotados, em um sistema de notas de uma a cinco estrelas, e uma terceira lista com os módulos que mais são chamados como dependências de outros módulos.

No restante deste capítulo, estudaremos três módulos muito bem cotados nas três listas: Async, Commander e Underscore. Além destes, veremos alguns outros no restante do livro.

Async, um módulo para gerenciamento eficiente de callbacks

A aplicação mostrada no Exemplo 2.11, no Capítulo 2, tem um padrão de código assíncrono, no qual cada função é chamada em sequência e passa seus resultados para a próxima. Esse encadeamento é interrompido somente na ocorrência de um erro. Existem diversos padrões de código

comuns em Node, embora alguns sejam meras variações de outros e nem todos usem a mesma terminologia.

O exemplo também demonstra outro fenômeno, um problema bastante sério quando se encadeia muitos callbacks: a chamada *pirâmide do sofrimento eterno* (em inglês, pyramid of doom), na qual os callbacks aninhados são empurrados progressivamente para a direita até desaparecerem da tela.

Um dos módulos mais comumente usados para domar esse efeito é o módulo Async, que substitui o padrão de código típico dos callbacks aninhados por uma versão muito mais linear e administrável. Dentre os padrões de código assíncronos que o Async suporta estão:

`waterfall`
: Padrão em cascata. As funções são chamadas em sequência e o resultado de cada uma é passado como item de um array para o último callback (esse padrão é chamado por outros padrões, como, por exemplo, o padrão seriado, `series`, e o padrão sequencial, `sequence`).

`series`
: Padrão seriado. As funções são chamadas em sequência e, opcionalmente, os resultados são passados como itens de um array à última função de callback.

`parallel`
: Padrão paralelo ou simultâneo. As funções são executadas em paralelo e, quando completadas, os resultados são passados como itens de um array para o último callback (embora o array resultante não seja parte do padrão de código em algumas interpretações do padrão paralelo).

`whilst`
: Padrão “faça enquanto”. Chama uma função repetidamente, invocando o último callback apenas se um teste preliminar devolver o valor `false` ou se ocorrer algum erro.

`queue`

Padrão fila. Chama as funções em paralelo até que um determinado limite de simultaneidade seja atingido, após o qual quaisquer novas funções aguardarão em fila até que uma das funções em execução seja encerrada.

`until`

Padrão "até que". Chama repetidamente uma função, invocando o último callback somente se um teste de pós-processamento retornar `false` ou se ocorrer um erro.

`auto`

Padrão automático. As funções são chamadas com base em requisitos predefinidos; cada função recebe os resultados do callback imediatamente anterior.

`iterator`

Padrão iterador. Cada função chama a próxima e é capaz de acessar individualmente o próximo iterador.

`apply`

Padrão "aplicar". Uma função de continuação, com argumentos previamente aplicados, combinada com outras funções de controle de fluxo.

`nextTick`

Chama o callback na próxima varredura de um laço de eventos – baseado no `process.nextTick` do Node.

O módulo Async também oferece funcionalidade para gerenciar coleções e tem até suas próprias variantes de `forEach`, `map` e `filter`, bem como outras funções utilitárias, incluindo algumas de *gerenciamento de memória*. Por enquanto, o que nos interessa são os recursos de controle de fluxo.

O Async tem um repositório no GitHub.

Instale o Async pelo npm. Se preferir instalá-lo globalmente, use a opção `-g`. Para atualizar as dependências, use `--save` ou `--save-dev`:

```
npm install async
```

O Async oferece controle de fluxo para diversos padrões de código assíncronos, como os já descritos `serial`, `parallel` e `waterfall`. Por exemplo, o padrão descrito no Capítulo 2 funcionaria com `waterfall`, portanto usaremos o método `async.waterfall`. No Exemplo 3.5, `async.waterfall` é usado para abrir e ler um arquivo de dados usando `fs.readFile`, fazer a substituição síncrona de strings e depois gravar a string no arquivo original usando `fs.writeFile`. Preste muita atenção à função de callback usada em cada passo.

Exemplo 3.5 – Usando async.waterfall para ler, modificar e escrever o conteúdo de um arquivo de forma assíncrona

```
var fs = require('fs'),
   async = require('async');

async.waterfall([
   function readData(callback) {
      fs.readFile('./data/data1.txt', 'utf8', function(err, data){
         callback(err,data);
      });
   },

   function modify(text, callback) {
      var adjdata=text.replace(/somecompany\.com/g,'burningbird.net');
      callback(null, adjdata);
   },
   function writeData(text, callback) {
      fs.writeFile('./data/data1.txt', text, function(err) {
      callback(err,text);
      });
   }
], function (err, result) {
      if (err) {
         console.error(err.message);
      } else {
         console.log(result);
      }
});
```

O método `async.waterfall` recebe dois parâmetros: um array de tarefas a serem executadas e uma função final de callback, que é opcional. Cada

função representando uma tarefa assíncrona é um elemento do array de `async.waterfall` e cada uma delas requer um callback como o último parâmetro de sua lista de parâmetros. É essa função de callback que nos permite encadear os resultados assíncronos dos callbacks sem ter de aninhar as funções fisicamente. Observe que, como fica claro no código, o callback de cada função é tratado da mesma maneira como seria normalmente tratado se usássemos callbacks aninhados – a única diferença é que não precisamos testar os erros em cada função. O Async procura um objeto de erro no primeiro parâmetro em cada callback. Se passarmos um objeto de erro no callback, o processo é encerrado nesse ponto e a rotina final de callback é chamada. Esse callback final é o momento em que podemos processar o resultado final ou, se existir, o erro.

O Exemplo 3.5 usa funções identificadas, embora na documentação do Async sejam mostradas apenas funções anônimas. O emprego de funções identificadas pode simplificar a depuração e o tratamento de erros, mas ambos os tipos de função resolvem o problema com a mesma qualidade.

O processamento é bastante semelhante ao que obtivemos no Capítulo 2, mas sem o aninhamento (e sem ter de testar explicitamente o erro em cada função). Parece complicado à primeira vista, e eu não recomendaria o emprego de Async quando o aninhamento tem poucos níveis, mas observe o que ele pode fazer quando o nível de aninhamento é maior. O Exemplo 3.6 duplica a funcionalidade que vimos no exemplo do Capítulo 2, mas sem aninhamento ou indentação excessivos.

Exemplo 3.6 – Lê objetos da pasta, testa se é um arquivo, lê o arquivo, modifica o conteúdo e o grava, mostrando os resultados no console de log

```
var fs = require('fs'),
   async = require('async'),
   _dir = './data/';

var writeStream = fs.createWriteStream('./log.txt',
   {'flags' : 'a',
    'encoding' : 'utf8',
    'mode' : 0666});

async.waterfall([
   function readDir(callback) {
```

```
      fs.readdir(_dir, function(err, files) {
         callback(err,files);
      });
   },
   function loopFiles(files, callback) {
      files.forEach(function (name) {
         callback (null, name);
      });
   },

   function checkFile(file, callback) {
      fs.stat(_dir + file, function(err, stats) {
         callback(err, stats, file);
      });
   },
   function readData(stats, file, callback) {
      if (stats.isFile())
         fs.readFile(_dir + file, 'utf8', function(err, data){
            callback(err,file,data);
         });
   },
   function modify(file, text, callback) {
      var adjdata=text.replace(/somecompany\.com/g,'burningbird.net');
      callback(null, file, adjdata);
   },
   function writeData(file, text, callback) {
      fs.writeFile(_dir + file, text, function(err) {
         callback(err,file);
      });
   },
   function logChange(file, callback) {
      writeStream.write('changed ' + file + '\n', 'utf8', function(err) {
         callback(err);
      });
   }
], function (err) {
      if (err) {
         console.error(err.message);
      } else {
         console.log('modified files');
      }
});
```

A funcionalidade é absolutamente idêntica à do exemplo do Capítulo 2.

O método `fs.readdir` é usado para obter um array com os objetos da pasta. O método `forEach` do Node (não o `forEach` do Async) é usado para acessar cada objeto específico. O método `fs.stats` é usado para obter os `stats` (dados estatísticos) de cada objeto. `stats` verifica se o objeto é um arquivo e, se for, abre-o e acessa seus dados. Esses dados são modificados e passados adiante para a função de gravação no arquivo, `fs.writeFile`. A operação é gravada no arquivo de log e cada operação que tenha sucesso mostra uma mensagem no console.

Observe que uma quantidade maior de dados é passada em alguns callbacks, em relação aos outros. Muitas das funções precisam do nome do arquivo e de seu conteúdo textual, portanto esses dados são passados na maioria dos últimos métodos. Qualquer quantidade de dados pode ser passada aos métodos, desde que o primeiro parâmetro seja um objeto de erro (ou `null`, se não houver erro) e o último parâmetro de cada função seja sua correspondente função de callback. Não precisamos verificar manualmente os erros em cada função assíncrona que executa uma das tarefas, porque o Async já testa automaticamente o objeto de erro em cada callback. Se um erro for encontrado, Async interrompe o processamento e chama o callback final.

Os outros métodos de controle de fluxo do Async, `async.parallel` e `async.serial`, funcionam de forma parecida, recebendo um array de tarefas no primeiro parâmetro e uma função opcional de callback do segundo. O que difere, como era de esperar, é a maneira como as tarefas assíncronas são processadas.

O método `async.parallel` chama todas as funções assíncronas de uma só vez e, quando todas tiverem terminado o que tinham que fazer, chama o callback final, que é opcional. O Exemplo 3.7 usa `async.parallel` para ler o conteúdo de três arquivos em paralelo. Contudo, em vez de um array de funções, este exemplo usa uma técnica alternativa, também suportada pelo Async: cada função assíncrona responsável por uma tarefa é passada como um objeto (e não como um array), sendo cada tarefa atribuída a uma propriedade desse objeto. Os resultados são enviados ao console quando as três tarefas forem completadas.

Exemplo 3.7 – Três arquivos são abertos em paralelo e têm seu conteúdo lido

```
var fs = require('fs'),
   async = require('async');

async.parallel({
   data1 : function (callback) {
      fs.readFile('./data/fruit1.txt', 'utf8', function(err, data){
         callback(err,data);
      });
   },
   data2 : function (callback) {
      fs.readFile('./data/fruit2.txt', 'utf8', function(err, data){
         callback(err,data);
      });
   },
   data3 : function readData3(callback) {
      fs.readFile('./data/fruit3.txt', 'utf8', function(err, data){
         callback(err,data);
      });
   },

}, function (err, result) {
      if (err) {
         console.log(err.message);
      } else {
         console.log(result);
      }
});
```

Os resultados são devolvidos como um array de objetos, com cada resultado associado a cada uma das propriedades. Em nosso exemplo, os três arquivos de dados têm o seguinte conteúdo:

- *fruit1.txt*: apples
- *fruit2.txt*: oranges
- *fruit3.txt*: peaches

O resultado, quando executamos o código do Exemplo 3.7, é:

```
{ data1: 'apples\n', data2: 'oranges\n', data3: 'peaches\n' }
```

Como exercício, tente testar você mesmo, sozinho, os outros métodos de controle de fluxo do Async. Lembre-se apenas de que, ao trabalhar com os métodos de controle de fluxo do Async, pouca coisa é necessária:

informar uma função de callback, que será chamada quando a tarefa terminar, para cada tarefa assíncrona; passar um objeto de erro (ou, se não houver erros possíveis, passar `null`); e, claro, quaisquer dados com os quais se queira trabalhar.

Commander: fazendo mágica no terminal

A ferramenta Commander simplifica a implementação de opções em nossa aplicação, que podem ser usadas quando chamamos a aplicação no terminal. Essas opções a que nos referimos são as "chaves" de ativação de funcionalidade da aplicação, como, por exemplo, a opção `-h` (ou `--help`), que mostra a página de ajuda.

O Commander deve ser instalado pelo npm:

```
npm install commander
```

Na aplicação ou módulo, deve ser incluído usando a instrução `require`:

```
var program = require('commander');
```

Para usá-lo, encadeie as possíveis opções, listando todas as que forem suportadas pela aplicação. No código a seguir, a aplicação suporta duas opções por default – `-V` (e sua variante `--version`), para mostrar a versão da aplicação que está instalada, e `-h` (ou `--help`) para mostrar um texto de ajuda – e implementa também duas opções personalizadas: `-s` ou `--source` para o site contendo o código-fonte e `-f` ou `--file` para informar um nome de arquivo.

```
var program = require('commander');

program
   .version ('0.0.1')
   .option ('-s, --source [web site]', 'Source web site')
   .option ('-f, --file [file name]', 'File name')
   .parse(process.argv);

console.log(program.source);
console.log(program.file);
```

Precisamos fornecer as opções personalizadas, enquanto o Commander cria sozinho as opções default de versão e ajuda. Ao rodar a aplicação:

```
node options -h
```

o Commander mostra na tela as opções disponíveis:

```
Usage: options [options]

Options:

  -h, --help output usage information
  -V, --version output the version number
  -s, --source [web site] Source web site
  -f, --file [file name] File name
```

As opções em formato abreviado podem ser concatenadas, como em `-sf`, o que é perfeitamente válido para o Commander. Ele também suporta opções com palavras encadeadas, como `--file-name`, e usa formatação CamelCase para produzir o resultado: `program.fileName`.

O Commander também suporta:

- Coerção (type casting):

  ```
  .option('-i, --integer <n>', 'An integer argument', parseInt)
  ```

- Expressões regulares:

  ```
  .option('-d --drink [drink]', 'Drink', /^(coke|pepsi|izze)$/i)
  ```

Outro recurso é o argumento *variadic* na última opção, o que abre a possibilidade de receber um número variável de argumentos. Por exemplo, imagine que a aplicação possa receber um número variável e arbitrário de palavras-chave. Para implementar isso usando o Commander, faça:

```
var program = require('commander');

program
   .version('0.0.1')
   .command('keyword <keywork> [otherKeywords...]')
   .action(function (keyword, otherKeywords) {
      console.log('keyword %s', keyword);
      if (otherKeywords) {
         otherKeywords.forEach(function (oKey) {
            console.log('keyword %s', oKey);
         });
      }
   });

program.parse(process.argv);
```

Se rodarmos a linha a seguir:

```
node options2 keyword one two three
```

teremos como resultado:

```
keyword one
keyword two
keyword three
```

Download do Commander

Para baixar o Commander diretamente, dirija-se a seu repositório no GitHub. O Commander é especialmente útil para aplicações que rodarão no terminal, como demonstraremos no Capítulo 6.

O onipresente Underscore

O módulo Underscore pode ser instalado com o comando:

```
npm install underscore
```

De acordo com seus criadores, o Underscore é um módulo de terceiros para o Node, oferecendo funcionalidades avançadas em JavaScript que costumeiramente encontramos em bibliotecas de frontend, como o jQuery ou o Prototype.js.

O Underscore foi batizado assim porque, tradicionalmente, o nome do módulo no código, pelo qual acessamos os métodos que descortinam sua funcionalidade, é um simples caractere de sublinhado, ou underscore[2], em inglês (_), semelhante ao que acontece com o jQuery, que é chamado pelo caractere `$`. Veja este exemplo:

```
var _ = require('underscore');
_.each(['apple','cherry'], function (fruit) { console.log(fruit); });
```

Obviamente, o problema de usar o caractere underscore é que ele tem um significado específico no REPL, como veremos no Capítulo 4. Não há problema, podemos substituir o underscore por uma variável com nome à nossa escolha – em nosso exemplo, `us`:

```
var us = require('underscore');
us.each(['apple','cherry'], function(fruit) { console.log(fruit); });
```

O Underscore oferece truques avançados para trabalhar com arrays, coleções, funções, objetos, aninhamento e alguns utilitários de uso geral. Felizmente, a documentação é farta e muito bem-feita, e está disponível

no site oficial do módulo, portanto vou me abster de detalhar seu uso aqui.

Entretanto, analisaremos um recurso muito bacana: uma maneira controlada de estender o Underscore com suas próprias funções internas usando uma em especial, `mixin`. Podemos testar este e outros métodos, rapidamente, em uma sessão REPL:

```
> var us = require('underscore');
> us.mixin({
... betterWithNode: function(str) {
..... return str + ' is better with Node';
..... }
... });
> console.log(us.betterWithNode('chocolate'));
chocolate is better with Node
```

Veremos o termo *mixin* ser usado em inúmeros módulos do Node. A funcionalidade é baseada em um padrão de código no qual as propriedades de um objeto são adicionadas ("mixadas") a outro.

O Underscore não é o único módulo utilitário com notas altas no ranking do npm. Dê uma olhada, por exemplo, no lodash. Estude bem esses dois módulos. O site do lodash traz documentação completa e de excelente qualidade.

---

1 N. do T.: Neste contexto, código arbitrário é qualquer trecho de código externo à aplicação ou aos módulos chamados por ela. Pode ser, por exemplo, código digitado pelo usuário em um campo de texto ou código enviado por outra aplicação para que seja processado pela sua.

2 N. do T.: O nome do caractere _ em inglês é underscore. No Brasil, é popular o uso incorreto de outra expressão inglesa, underline, para nomeá-lo; um erro muito comum, embora grosseiro. A palavra underline, como substantivo, não existe na língua inglesa. A expressão em português "caractere de sublinhado", apesar de constar de grande número de glossários, praticamente não é usada na literatura nacional. Resumindo: é errado falar underline, não use essa palavra!

CAPÍTULO 4

Tornando o Node interativo com o REPL e o poder do console

Para explorar o potencial do Node e aprender como criar código para nossas aplicações ou módulos, não é necessário digitar o código em JavaScript num arquivo e executá-lo com o Node toda vez que quisermos testar algo. O Node vem com um componente interativo conhecido como REPL, sigla que significa *read-eval-print loop* ou, em português, *laço de leitura-cálculo-exibição*.

O REPL (pronuncia-se "répou") suporta a edição simplificada de linha e um pequeno conjunto de comandos básicos. Qualquer coisa digitada no REPL é, na maioria das vezes, processada de forma idêntica à que seria se fosse gravada num arquivo e executado usando o Node. Podemos até mesmo usar o REPL para criar o código completo da aplicação – literalmente testando-a no exato momento em que é digitada.

Neste capítulo, veremos como usar o REPL, bem como algumas de suas deficiências e como contorná-las. Os artifícios usados para desviar dessas deficiências incluem substituir o mecanismo subjacente de persistência de comandos e empregar edição direta na linha digitada. Se o REPL nativo não entregar exatamente o ambiente interativo de que precisamos, há até uma API para que possamos criar nosso próprio REPL personalizado.

O REPL é uma das ferramentas essenciais para o desenvolvimento em Node, e o mesmo se pode dizer do console, que usaremos na maioria das aplicações mostradas no livro. Contudo, tenha em mente que o console tem muito mais recursos e utilidade, e usá-lo apenas para mostrar mensagens de log é desperdício.

REPL: primeiras impressões e expressões indefinidas

Para abrir o REPL, simplesmente digite `node` no terminal sem informar o nome de nenhuma aplicação:

```
$ node
```

O REPL mostra um prompt de comando – um sinal de “maior que” (>). Qualquer coisa digitada daqui em diante é processada pela máquina JavaScript do Node, a V8.

Usar o REPL é extremamente simples, basta digitar o código em JavaScript desejado:

```
> a = 2;
2
```

A ferramenta mostra imediatamente o resultado da expressão digitada. No exemplo, o valor calculado da expressão é `2`. No exemplo a seguir, o resultado é um array com três elementos:

```
> b = ['a','b','c'];
[ 'a', 'b', 'c' ]
```

Para acessar a última expressão, use a variável especial underscore (_). No exemplo a seguir, `a` recebe o valor `2`, e a expressão resultante é incrementada de 1, e depois de mais 1:

```
> a = 2;
2
> ++_;
3
> ++_;
4
```

Podemos até acessar propriedades ou chamar métodos na expressão armazenada no underscore:

```
> ['apple','orange','lime']
[ 'apple', 'orange', 'lime' ]
> _.length
3
> 3 + 4
7
> _.toString();
'7'
```

Podemos usar a palavra-chave `var` com o REPL para acessar uma expressão ou valor mais tarde, mas tenha em mente que podem ser produzidos resultados inesperados. Por exemplo, considere a linha a seguir no REPL:

```
var a = 2;
```

Se examinarmos a variável, não obteremos o valor `2`; em vez disso, teremos `undefined`. A razão para tal é que o resultado da expressão é indefinido visto que a variável não retorna um resultado definido quando é avaliada[1].

Considere, em vez disso, o exemplo a seguir, que mostra mais ou menos como as coisas funcionam internamente no REPL:

```
console.log(eval('a = 2'));
console.log(eval('var a = 2'));
```

Ao digitar as duas linhas em um arquivo e as rodar no Node como uma aplicação, temos:

```
2
undefined
```

Não há resultado para o segundo `eval`; por isso, o valor devolvido é `undefined`. O REPL é um laço de leitura-cálculo-exibição (read-eval-print loop), com ênfase no *cálculo* (*eval*).

Podemos usar a variável no REPL, da mesma forma como faríamos numa aplicação em Node:

```
> var a = 2;
undefined
> a++;
2
> a++;
3
```

Os dois últimos comandos têm resultado, e são exibidos pelo REPL.

Veremos como criar um REPL personalizado – um que não mostra `undefined` – em “Um REPL para chamar de seu”.

Para encerrar a sessão do REPL, pressione Ctrl-C duas vezes, ou Ctrl-D uma vez. Veremos outras formas de encerrar a sessão em “Comandos do REPL”.

Benefícios do REPL: entendendo o JavaScript por debaixo dos panos

O exemplo a seguir é uma demonstração típica do REPL:

```
> 3 > 2 > 1;
false
```

Esse trecho de código mostra muito bem como o REPL pode ser útil. À primeira vista, podemos esperar que a expressão deva ser avaliada como `true`, pois 3 é maior que 2, por sua vez maior que 1. Contudo, em JavaScript, os operadores relacionais são avaliados da esquerda para a direita, e o resultado de cada expressão é enviado à próxima expressão a ser avaliada.

Uma maneira fácil de visualizar o que aconteceu com a expressão anterior é esta sessão do REPL:

```
> 3 > 2 > 1;
false
> 3 > 2;
true
> true > 1;
false
```

Agora o resultado faz mais sentido. O que acontece é que a expressão 3 > 2 é avaliada e produz o resultado `true`. Logo em seguida, o valor `true` (que é booleano) é comparado com o valor numérico 1. O JavaScript faz a conversão automática de tipos, e decorre disso que `true` e 1 são valores booleanos equivalentes. Ora, `true` não é maior que 1, portanto o resultado da expressão é, de fato, `false`.

A conveniência do REPL nos permite descobrir esses pequenos problemas interessantes, tanto em nosso código quanto no próprio JavaScript. Testando o código no REPL, nos certificamos de não produzir efeitos colaterais em nossas aplicações (como, por exemplo, esperar um resultado `true` de uma expressão falsa, que "teima" em nos devolver `false`).

JavaScript mais complexo e com várias linhas

Podemos digitar no REPL o mesmo JavaScript que escreveríamos em um

arquivo, incluindo instruções `require` para importar módulos. No exemplo a seguir, mostramos uma sessão de uso do REPL com o intuito de experimentar o módulo Query String (`qs`):

```
$ node
> qs = require('querystring');
{ unescapeBuffer: [Function],
  unescape: [Function],
  escape: [Function],
  encode: [Function],
  stringify: [Function],
  decode: [Function],
  parse: [Function] }
> val = qs.parse('file=main&file=secondary&test=one').file;
[ 'main', 'secondary' ]
```

Como não empregamos a palavra-chave `var`, o resultado da expressão é mostrado – nessa instância, é a interface para o objeto `querystring`. Bacana, não? Não apenas temos acesso ao objeto como também aprendemos mais sobre sua interface. Entretanto, se quisermos evitar a exibição de todo esse texto, que pode ser bastante longo, aliás, basta usar a palavra-chave `var`:

```
> var qs = require('querystring');
```

Ambas as formas permitirão acesso ao objeto `querystring` por meio da variável `qs`.

Além de sermos capazes de incorporar módulos externos, o REPL sabe lidar maravilhosamente com expressões que precisam de mais de uma linha, desde que o código seja aninhado entre chaves ({}):

```
> var test = function (x, y) {
... var val = x * y;
... return val;
... };
undefined
> test(3,4);
12
```

O REPL mostra reticências para indicar que tudo o que está sendo digitado veio depois da abertura das chaves e que o comando ainda não foi finalizado com uma chave de fechamento. O mesmo vale para parênteses ainda abertos:

```
> test(4,
... 5);
20
```

Quanto maior o nível de aninhamento, maior a quantidade de pontos. Isso é absolutamente necessário em um ambiente interativo como o REPL, pois do contrário seria muito fácil perder de vista o nível em que estamos à medida que o código é digitado:

```
> var test = function (x, y) {
... var test2 = function (x, y) {
..... return x * y;
..... }
... return test2(x,y);
... }
undefined
> test(3,4);
12
>
```

Podemos digitar e executar uma aplicação inteira no REPL e até mesmo copiar e colar essa aplicação a partir de um arquivo. No exemplo a seguir, suprimi os valores reais de exibição do servidor, mostrados em negrito, porque são muito longos – e porque, provavelmente, já serão bem diferentes quando este livro chegar às mãos do leitor:

```
> var http = require('http');
undefined
> http.createServer(function (req, res) {
...
... // cabeçalho de conteúdo
... res.writeHead(200, {'Content-Type': 'text/plain'});
...
... res.end("Hello person\n");
... }).listen(8124);
{ domain:
   { domain: null,
      _events: { error: [Function] },
      _maxListeners: undefined,
      members: [] },
   _
...
   httpAllowHalfOpen: false,
```

```
    timeout: 120000,
    _pendingResponseData: 0,
    _connectionKey: '6:null:8124' }
> console.log('Server running at http://127.0.0.1:8124/');
Server running at http://127.0.0.1:8124/
Undefined
```

Essa aplicação, que está rodando direto no REPL, pode ser acessada pelo navegador como se fosse uma legítima aplicação rodando em Node, a partir de um arquivo.

A "desvantagem" de não atribuir uma expressão à variável é que obteremos na tela uma longa lista com as propriedades do objeto no meio de nossa aplicação. Entretanto, um dos meus usos favoritos do REPL é ter uma visão geral de todos os objetos usados. Por exemplo, o objeto nativo `global` é documentado de forma um tanto espalhada no site do Node.js. Para ter um resumo (bastante explicativo) desse objeto, abri uma sessão do REPL e passei o objeto para o método `console.log` mais ou menos assim:

```
> console.log(global)
```

Eu poderia ter feito o mesmo usando uma atribuição a uma variável, como no exemplo a seguir, e teria resultados muito semelhantes. De quebra, teria a variável `gl`, com a qual já poderia trabalhar. O seguinte resultado foi resumido por questão de espaço:

```
> gl = global;
...
_: [Circular],
gl: [Circular] }
```

Uma simples chamada ao objeto `global` produz o mesmo efeito:

```
> global
```

Não vou reproduzir aqui tudo o que é mostrado no REPL, pois a interface do objeto `global` é muito grande. Peço que o leitor experimente em sua própria instalação e veja por si mesmo. A lição mais importante a ser aprendida com este exercício é que podemos, a qualquer tempo, rápida e facilmente olhar a interface de um objeto. É uma maneira fácil de relembrar quais métodos podem ser chamados e quais propriedades estão disponíveis.

Discutimos bastante sobre o `global` no Capítulo 2.

Podemos usar as setas do teclado para navegar pelos comandos anteriores já digitados no REPL, algo bastante útil para revisar e editar o que já foi feito, mesmo que de forma limitada.

Considere a seguinte sessão no REPL:

```
> var myFruit = function(fruitArray,pickOne) {
... return fruitArray[pickOne - 1];
... }
undefined
> fruit = ['apples','oranges','limes','cherries'];
[ 'apples', 'oranges', 'limes', 'cherries' ]
> myFruit(fruit,2);
'oranges'
> myFruit(fruit,0);
undefined
> var myFruit = function(fruitArray,pickOne) {
... if (pickOne <= 0) return 'invalid number';
... return fruitArray[pickOne - 1];
... };
undefined
> myFruit(fruit,0);
'invalid number'
> myFruit(fruit,1);
'apples'
```

Embora não esteja demonstrado nessa listagem, quando modificamos a função para verificar o valor de entrada, o que fiz foi usar a seta para cima até a declaração da função, modificá-la e pressionar Enter para reiniciar. Adicionei a nova linha e novamente usei as setas do teclado para repetir linhas digitadas anteriormente até que a função estivesse como eu queria. Também usei as setas para repetir a chamada à função que resultou em `undefined`.

Pode parecer trabalho demais só para não repetir a digitação, mas imagine que estejamos trabalhando com expressões regulares, como estas:

```
> var ssRe = /^\d{3}-\d{2}-\d{4}$/;
undefined
> ssRe.test('555-55-5555');
```

```
true
> var decRe = /^\s*(\+|-)?((\d+(\.\d+)?)|(\.\d+))\s*$/;
undefined
> decRe.test(56.5);
true
```

Sou absolutamente incompetente quando se trata de expressões regulares e preciso mexer nelas dezenas de vezes até que estejam funcionando do jeito que eu quero. Com o REPL, posso testá-las facilmente. Todavia, redigitá-las – especialmente as longas – a cada teste pode se mostrar um trabalho monstruosamente grande.

Por sorte, tudo o que temos de fazer no REPL é usar as setas do teclado para encontrar a linha em que está a expressão regular, editá-la, pressionar Enter e continuar os testes.

Além das teclas de direção, a tecla Tab pode ser usada para *autocompletar* um comando, desde que não haja confusão sobre o que está sendo autocompletado. Por exemplo, digite **`va`** na linha de comando e pressione Tab. O REPL listará duas instruções possíveis: `var` e `valueOf`, pois ambas podem completar o que foi digitado. Por outro lado, digitar `querys` seguido de Tab devolve a única opção disponível, `querystring`. Podemos também usar a tecla Tab para autocompletar qualquer variável global ou local. A Tabela 4.1 nos dá um resumo dos comandos de teclado que funcionam com o REPL.

Tabela 4.1 – Controle de teclado no REPL

Atalho de teclado	O que ele faz
Ctrl-C	Encerra o comando atual. Pressionar Ctrl-C duas vezes força o encerramento.
Ctrl-D	Sai do REPL.
Tab	Autocompleta uma variável global ou local.
Seta para cima	Navega pelo histórico de comandos em direção ao mais antigo.
Seta para baixo	Navega pelo histórico de comandos em direção ao mais novo.
Underscore (_)	Referencia o resultado da última expressão.

Uma preocupação bastante válida é o destino de tudo o que digitamos no REPL. Podemos passar horas digitando nele e depois perder tudo. Não se preocupe com isso: é possível salvar em um arquivo todos os resultados

do contexto atual usando o comando `.save`. Este e outros comandos do REPL serão abordados nas próximas páginas.

Comandos do REPL

O REPL mostra uma interface simples, com um conjunto pequeno de comandos muito úteis. Na seção anterior, mencionamos o comando `.save`, que salva em um arquivo todas as entradas no contexto atual do objeto. A menos que um novo contexto de objeto seja criado, ou que o comando `.clear` seja emitido, o contexto deve encampar todas as linhas digitadas na sessão atual do REPL:

```
> .save ./dir/session/save.js
```

Somente o que foi digitado é salvo, como se tivesse sido digitado diretamente em um arquivo, usando um editor de textos.

A lista a seguir mostra todos os comandos do REPL e sua finalidade:

`.break`
: Se nos perdermos durante a digitação de uma mesma linha de código que foi desmembrada em várias pelo REPL porque não coube na tela, basta digitar `.break` para limpar tudo e iniciar em uma linha vazia. Tudo o que já foi digitado será perdido.

`.clear`
: Reinicia o objeto de contexto e limpa todas as expressões que ocupem mais de uma linha. Basicamente, esse comando apaga tudo e reinicia em um contexto completamente limpo.

`.exit`
: Sai do REPL.

`.help`
: Mostra todos os comandos disponíveis no REPL.

`.save`
: Salva a sessão atual do REPL em um arquivo.

`.load`
: Abre um arquivo e carrega seu conteúdo na sessão atual (`.load /caminho/para/file.js`).

Se estiver trabalhando em uma aplicação e usando o REPL como editor, uma dica: salve seu trabalho com bastante frequência usando o comando `save`. Embora o histórico completo dos comandos digitados persista na memória, tentar recriar seu código a partir do histórico será uma experiência regada a choro e ranger de dentes.

E por falar em histórico e persistência, veremos a seguir como personalizá-los no REPL.

REPL e rlwrap

A documentação do REPL disponível no site do Node.js menciona a criação de uma variável de ambiente para que possamos usar o REPL com o `rlwrap`. Contudo, o leitor poderia perguntar o que seria esse `rlwrap` e por que ele deve ser usado com o REPL?

O utilitário `rlwrap` é um wrapper que adiciona a funcionalidade da biblioteca GNU `readline` a comandos do terminal, o que promove grande flexibilidade quando trabalhamos com o teclado. A biblioteca intercepta os dados vindos do teclado e oferece funcionalidade adicional, como, por exemplo, edição avançada na própria linha digitada e um histórico persistente de comandos.

É preciso instalar tanto o `rlwrap` quanto o `readline` para poder usufruir deles no REPL. Muitos sabores de Unix facilitam a instalação deles em seus sistemas de administração de pacotes de software. Por exemplo, no Ubuntu, instalar o `rlwrap` resume-se a:

```
apt-get install rlwrap
```

Os usuários de Mac devem usar o instalador apropriado para esses aplicativos. No Windows, é necessário usar um emulador de ambiente Unix, como o Cygwin.

A linha a seguir é um exemplo rápido e visual do poder que emana da combinação REPL e `rlwrap`. Aqui, alteramos para lilás, com um único

comando, a cor do prompt do REPL:

```
NODE_NO_READLINE=1 rlwrap -ppurple node
```

Para tornar essa mudança definitiva, basta adicionar um alias ao arquivo *.bashrc* do usuário:

```
alias node="NODE_NO_READLINE=1 rlwrap -ppurple node"
```

Para mudar não só a cor, mas também o formato do prompt, o comando é:

```
NODE_NO_READLINE=1 rlwrap -ppurple -S "::> " node
```

A partir de agora, o prompt do Node será lilás e se parecerá com isto:

```
::>
```

Há um componente especialmente útil no `rlwrap`, que habilita a persistência do histórico entre sessões distintas do REPL. Por default, o histórico vale apenas para aquela sessão do REPL. Fechou, perdeu. Usando o `rlwrap`, na próxima vez que usarmos o REPL teremos acesso não apenas ao histórico de comandos da sessão atual, mas também ao das sessões anteriores (bem como qualquer outra coisa digitada na linha de comando). No exemplo de sessão a seguir, os comandos mostrados não foram digitados, mas recuperados do histórico com a tecla de seta para cima, *depois* de eu ter encerrado o REPL e entrado nele novamente:

```
::> e = ['a','b'];
[ 'a', 'b' ]
::> 3 > 2 > 1;
false
```

Por mais útil que seja o `rlwrap`, ainda acabamos recebendo um `undefined` toda vez que digitamos uma expressão que não retorna valor. Entretanto, esse comportamento (e outros mais) é ajustável, basta criar nosso próprio REPL personalizado, como discutiremos a seguir.

Um REPL para chamar de seu

O Node oferece uma API com a qual podemos criar um REPL personalizado. Para isso, a primeira coisa a fazer é incluir o módulo do REPL (`repl`):

```
var repl = require("repl");
```

Para criar um novo REPL, chamamos o método `start` no objeto `repl`. A sintaxe para esse método é:

```
repl.start(options);
```

O objeto `options` pode receber inúmeros valores. Os mais importantes para nós, no momento, são:

`prompt`

O default é `>`.

`input`

Fluxo de dados de leitura. O default é `process.stdin`.

`output`

Fluxo de dados de escrita. O default é `process.stdout`.

`eval`

O default é um wrapper `async` para o `eval`.

`useGlobal`

O default é `false`, o que inicia um novo contexto em vez de usar o objeto global.

`useColors`

Decide se a função `writer` deve usar cores. O default é o valor `terminal` do REPL.

`ignoreUndefined`

O default é `false`: não ignora as respostas `undefined`.

`terminal`

Com o valor `true`, o fluxo de dados será tratado como um tty (terminal), incluindo suporte a códigos de escape ANSI/VT100.

`writer`

Função que avalia cada comando e devolve a formatação apropriada. O default é `util.inspect`.

`replMode`

Decide se o REPL roda em strict mode, default ou hybrid.

A partir do Node 5.8.0, `repl.start()` não mais obriga o uso de um objeto `options`.

Não acho muito edificante mostrar `undefined` toda vez que uma expressão não tiver um valor para devolver, portanto resolvi criar meu próprio REPL. Aproveitei para redefinir o prompt e travar o modo de operação em strict, portanto toda linha digitada é executada sob uma política “use strict”.

```
var repl = require('repl');

repl.start( {
   prompt: 'my repl> ',
   replMode: repl.REPL_MODE_STRICT,
   ignoreUndefined: true,
});
```

Rodei o arquivo, que chamei de *repl.js*, usando o Node:

```
node repl
```

Posso usar o meu próprio REPL da mesma forma que a versão nativa, mas agora tenho um prompt diferente e não sou mais importunado com a irritante mensagem de `undefined` depois da primeira atribuição de variável. Ainda recebo qualquer outra mensagem que não seja `undefined`:

```
my repl> let ct = 0;
my repl> ct++;
0
my repl> console.log(ct);
1
my repl> ++ct;
2
my repl> console.log(ct);
2
```

No código mostrado, todos os valores default do REPL continuam os mesmos, os únicos discordantes foram os que alterei explicitamente. Deixar de listar as outras propriedades do objeto `options` faz com que as não listadas usem os valores default.

Podemos substituir a função `eval` em nosso REPL personalizado. Atente

para o formato específico:

```
function eval(cmd, callback) {
   callback(null, result);
}
```

As opções de `input` e `output` são interessantes. Podemos executar inúmeras versões do REPL, recebendo dados tanto da entrada-padrão (o default) quanto de sockets. A documentação do REPL no site do Node.js mostra um exemplo de um REPL “escutando” um socket TCP usando o código a seguir:

```
var net = require("net"),
    repl = require("repl");

connections = 0;

repl.start({
   prompt: "node via stdin> ",
   input: process.stdin,
   output: process.stdout
});

net.createServer(function (socket) {
   connections += 1;
   repl.start({
      prompt: "node via Unix socket> ",
      input: socket,
      output: socket
   }).on('exit', function() {
      socket.end();
   })
}).listen("/tmp/node-repl-sock");

net.createServer(function (socket) {
   connections += 1;
   repl.start({
      prompt: "node via TCP socket> ",
      input: socket,
      output: socket
   }).on('exit', function() {
      socket.end();
   });
}).listen(5001);
```

Ao executar a aplicação, aparece o prompt-padrão indicando onde a

aplicação em Node está rodando. Entretanto, podemos acessar o REPL também via TCP. É possível usar o PuTTY como cliente de Telnet para acessar essa versão do REPL. Funciona... até certo ponto. Foi preciso emitir primeiro um comando `.clear`, a formatação fica toda errada, e quando tentamos usar o caractere underscore para referenciar a última expressão, o Node não o reconheceu. Tentei também com o cliente de Telnet nativo do Windows, e os resultados foram ainda piores. Entretanto, se usarmos o cliente de Telnet que vem com o Linux, tudo funciona às mil maravilhas[2].

O problema aqui, como era de esperar, são as configurações do cliente Telnet. Como rodar o REPL a partir de um socket Telnet não está nos meus planos, deixei por isso mesmo. Não recomendo usar o Telnet para nada, pelo alto risco de segurança envolvido. A título de comparação, é muitíssimo mais inseguro do que usar `eval()` em código a ser executado no navegador e não filtrar o texto digitado pelo usuário antes de entregá-lo à função.

Podemos manter um REPL rodando e nos comunicar com ele via socket Unix usando o GNU Netcat:

```
nc -U /tmp/node-repl-sock
```

Digitar comandos nesse socket funciona do mesmo modo como se estivéssemos na entrada-padrão `stdin`. Contudo, tenha em mente que quando usamos um socket, seja TCP ou Unix, as mensagens geradas por `console.log` são mostradas no console do servidor, não no cliente:

```
console.log(someVariable); // Isso é mostrado no servidor,
                           // não no cliente!
```

Uma opção de aplicação que considero bastante útil é a criação de um REPL que, de antemão, carrega módulos. Na aplicação mostrada no Exemplo 4.1, depois que o REPL é iniciado, os módulos de terceiros Request (um cliente HTTP bastante poderoso), Underscore (um verdadeiro cinto de utilidades) e Q (gerenciamento de promises[3]) são carregados e atribuídos a propriedades do contexto.

Exemplo 4.1 – Um REPL personalizado que carrega módulos na inicialização

```
var repl = require('repl');
var context = repl.start({prompt: '>> ',
                          ignoreUndefined: true,
                          replMode: repl.REPL_MODE_STRICT}).context;
// carrega módulos
context.request = require('request');
context.underscore = require('underscore');
context.q = require('q');
```

Ao rodar a aplicação com o Node, obtemos o prompt do REPL e, nele, conseguimos acessar os módulos:

```
>> request('http://blipdebit.com/phoenix5a.png')
.pipe(fs.createWriteStream('bird.png'))
```

Os módulos nativos do Node não precisam ser incluídos especificamente, basta acessá-los diretamente pelo nome do módulo.

Para rodar a aplicação REPL como um executável no Linux, adicione a linha a seguir como primeira linha do script (o caminho pode variar de acordo com a distribuição):

```
#!/usr/local/bin/node
```

Modifique o arquivo da aplicação para executável:

```
$ chmod u+x replcontext.js
$ ./replcontext.js
>>
```

Acidentes acontecem – salve com frequência

O REPL do Node é uma ferramenta interativa que facilita bastante nossas tarefas de desenvolvimento. O REPL permite não apenas fazer experiências com o JavaScript antes de incluir o código em nossos arquivos: de fato, também podemos criar nossas aplicações interativamente e salvar os resultados quando tivermos terminado.

Outro recurso bastante útil do REPL é a criação de REPLs personalizados para, por exemplo, eliminar a hedionda mensagem de `undefined`, pré-carregar módulos, alterar o prompt e até mesmo mudar o comportamento de `eval` e outras coisas mais.

Também recomendo com veemência usar o REPL em conjunto com `rlwrap`

para que haja persistência de comandos entre sessões, o que acaba se mostrando um estupendo economizador de tempo. Além disso, quem dentre nós não aprecia ter à disposição alguns recursos avançados de edição?

À medida que exploramos o REPL mais a fundo, descobrimos que o velho ditado dos programadores de antigamente continua válido: acidentes acontecem. Salve suas coisas regularmente.

No momento em que decidimos usar o REPL para desenvolver nosso código, passando horas a fio sentado em frente ao terminal, e mesmo empregando o `rlwrap` para persistência do histórico, é necessário salvar o trabalho com frequência. Trabalhar com o REPL não é diferente do que trabalhar com outros ambientes de edição. Portanto, vou repetir: *acidentes acontecem – salve com frequência*.

O onipresente console

Poucos exemplos neste livro não fazem uso do console. O console é um canal no qual podemos mostrar valores, verificar o resultado de operações, verificar a natureza assíncrona de uma aplicação e providenciar um certo nível de sinalização.

Na maioria dos casos, usamos `console.log()` para mostrar mensagens. Mas o console é muito mais do que a versão para servidor de um alert box do navegador.

Tipos de mensagem no console, a classe Console e o bloqueio de processamento

Em muitos exemplos deste livro, usamos `console.log()` porque queremos que a aplicação nos avise a respeito de eventos e valores enquanto estamos experimentando com o Node. Essa função envia a mensagem para a saída-padrão, `stdout`, que tipicamente é o terminal. Contudo, ao criar aplicações em Node já visando o ambiente de produção será bastante útil usar outras funções de mensagem do console.

A função `console.info()` é equivalente a `console.log()`. Ambas escrevem em

`stdout` e ambas incluem um caractere de nova linha como parte da mensagem. A função `console.error()` difere apenas pelo fato de que a mensagem (sempre com um caractere de nova linha incluso) vai para `stderr`, não para `stdout`:

```
console.error("Ocorreu um erro...");
```

A função `console.warn()` tem o mesmo propósito.

Ambos os tipos de mensagem aparecem no terminal, então qual é a diferença? De fato, não há diferença. Para entender, precisamos olhar de perto o objeto `console`.

O módulo Logging

Não estamos limitados ao objeto nativo `console` para tarefas de registro de log. Há ferramentas mais sofisticadas, como os módulos de Bunyan e Winston.

Em primeiro lugar, o objeto `console` é global e instanciado na classe `Console`. Caso necessário, podemos criar nossa própria versão de `console` usando a mesma classe. Há duas maneiras de fazer isso.

Há duas maneiras de criar outra instância de `Console`, precisamos importar a classe `Console` ou acessá-la pelo objeto global `console`. Ambas resultam em um novo objeto que se parece com um console:

```
// usando require
var Console = require('console').Console;

var cons = new Console(process.stdout, process.stderr);
cons.log('testing');

// usando o objeto console já existente
var cons2 = new console.Console(process.stdout, process.stderr);

cons2.error('test');
```

Observe que, nas duas instâncias, as propriedades `process.stdout` e `process.stderr` são passadas como fluxos de escrita que recebem mensagens de log e de erro, respectivamente. O objeto global `console` é criado exatamente dessa maneira.

Aprendemos sobre `process.stdout` e `process.stderror` no Capítulo 2. O que sabemos sobre ambos é que entregam tudo o que é dirigido a eles para os arquivos descritores `stdout` e `stderr`, que pertencem ao sistema operacional, e que são diferentes da maioria dos fluxos de dados no Node, pois eles

são, tipicamente, blocantes – ou seja, síncronos. A única vez que não são síncronos é quando os fluxos são direcionados em um *pipe*. Em grande parte, o objeto `console` é blocante tanto para `console.log()` quanto para `console.error()`. Contudo, isso não chega a apresentar problemas, a não ser que estejamos redirecionando pelo fluxo uma quantidade muito grande de dados.

Por que, então, usar `console.error()` quando ocorre um erro? Porque em ambientes em que os dois fluxos são diferentes queremos garantir que o comportamento seja o mais correto. Se estivermos em um ambiente no qual as mensagens de log não sejam blocantes, mas as de erro são, queremos garantir que os erros realmente bloqueiem o processamento. Além disso, quando executamos uma aplicação em Node, podemos redirecionar as saídas de `console.log()` e `console.error()` para arquivos diferentes usando redirecionamento de linha de comando. O comando a seguir redireciona todas as mensagens de `console.log()` para um arquivo de logs e todos os erros para um arquivo separado:

```
node app.js 1> app.log 2> error.log
```

A aplicação:

```
// mensagens de log
console.log('this is informative');
console.info('this is more information');

// mensagens de erro
console.error('this is an error');
console.warn('but this is only a warning');
```

envia as duas primeiras linhas para *app.log* e as próximas duas para *error.log*.

Para voltar à classe `Console`, podemos duplicar a funcionalidade do objeto global `console` usando a classe `Console` e passando a elas `process.stdout` e `process.stderr`. Podemos ainda criar mais um objeto de console que redireciona a saída para fluxos diferentes, como arquivos separados para log e erros. A documentação da classe `Console` incluída pela Node Foundation traz este exemplo:

```
var output = fs.createWriteStream('./stdout.log');
var errorOutput = fs.createWriteStream('./stderr.log');
```

```
// log personalizado simples na variável logger
var logger = new Console(output, errorOutput);
// use-o como se fosse um console
var count = 5;
logger.log('count: %d', count);
// em stdout.log: count 5
```

A vantagem de usar esse tipo de objeto é que podemos usar o `console` global para mensagens de cunho geral, deixando o novo objeto para relatórios mais formais.

Processos e fluxos

O objeto `process` foi explicado no Capítulo 2 e os fluxos serão discutidos no Capítulo 6.

Formatando a mensagem com util.format() e util.inspect()

Todas as quatro funções de `console`, `log()`, `warn()`, `error()` e `info()`, podem receber qualquer tipo de dados, incluindo, claro, um objeto. Valores que não são objetos e não são strings são transformados por coerção em string. Entretanto, se o dado recebido for um objeto, saiba que o Node só consegue mostrar dois níveis de aninhamento. Se mais níveis forem necessários, use `JSON.stringify()` no objeto, o que resulta em uma árvore indentada muito mais legível:

```
var test = {
   a : {
      b : {
         c : {
            d : 'test'
         }
      }
   }
}

// apenas dois níveis de aninhamento são mostrados
console.log(test);

// três níveis de aninhamento são mostrados
var str = JSON.stringify(test, null, 3);

console.log(str);
```

A saída da aplicação é:

```
{ a: { b: { c: [Object] } } }
{
   "a": {
      "b": {
         "c": {
            "d": "test"
         }
      }
   }
}
```

Se o valor for uma string, podemos empregar a formatação no estilo printf para todas as quatro funções:

```
var val = 10.5;
var str = 'a string';

console.log('The value is %d and the string is %s', val, str);
```

Esse valor é vantajoso se estivermos trabalhando com dados passados como argumentos de uma função ou coletados de uma requisição web. O tipo de formatação permitida é baseado nas opções suportadas pelo módulo `util.format()`, que pode inclusive ser usado diretamente para criar a string:

```
var util = require('util');

var val = 10.5,
    str = 'a string';

var msg = util.format('The value is %d and the string is %s',val,str);
console.log(msg);
```

Embora estejamos usando a função diretamente, é mais simples empregá-la para formatar a mensagem em `console.log()`. Os valores de formatação permitidos são:

`%s`

string

`%d`

número (integer ou float, é indiferente)

`%j`

JSON. Substituído por ['circular'] se os argumentos contiverem

referências circulares

`%%`

mostra o caractere de porcentagem (%)

Argumentos extras são convertidos em string e concatenados na saída. Se o número de argumentos for menor que o esperado, o símbolo de formatação é impresso no lugar do valor faltante:

```
var val = 3;

// resulta em 'val é 3 e str é %s'
console.log('val é %d e str é %s', val);
```

As quatro funções mostradas neste capítulo não são as únicas usadas para informar coisas. Há também `console.dir()`.

A função `console.dir()` difere das outras pelo fato de que qualquer objeto que receba é repassado a `util.inspect()`. Essa função do módulo Utilities oferece um controle mais finito sobre como o objeto é mostrado, por meio de um objeto `options` secundário. Assim como `util.format()`, também pode ser usado diretamente. Por exemplo:

```
var test = {
   a : {
      b : {
         c : {
            d : 'test'
         }
      }
   }
}
var str = require('util').inspect(test, {showHidden: true, depth: 4 });
console.log(str);
```

O objeto é inspecionado e o resultado é devolvido como uma string baseada no que é recebido a partir do objeto `options`. As opções são:

- `showHidden` – Mostrar propriedades simbólicas ou não enumeráveis (o default é `false`).
- `depth` – Quantas vezes executar uma recursão para inspecionar o objeto (o default é `2`).
- `colors` – Se for `true`, a saída é estilizada com códigos de cores ANSI (o

default é `false`).

- `customInspect` – Se for `false`, as funções personalizadas de inspeção definidas nos objetos sendo inspecionados não serão chamadas (o default é `true`).

A mapeamento de cores pode ser definido globalmente usando um objeto `util.inspect.styles`. Podemos modificar as cores globais, também. Use `console.log()` para mostrar as propriedades do objeto:

```
var util = require('util');

console.log(util.inspect.styles);
console.log(util.inspect.colors);
```

A aplicação no Exemplo 4.2 modifica o objeto sendo mostrado para incluir uma data, um número e um valor lógico booleano. O código de cor do valor booleano é alterado de amarelo para azul para diferenciá-lo dos números comuns (por default, são ambos amarelos). O objeto é mostrado usando métodos variados: depois de ser processado por `util.inspect()`, usando `console.dir()` com as mesmas opções, usando a função básica `console.log()` e usando a função `JSON.stringify()` no objeto.

Exemplo 4.2 – Diferentes opções de formatação para exibir um objeto na tela

```
var util = require('util');

var today = new Date();

var test = {
   a : {
      b : {
         c : {
            d : 'test'
         },
         c2 : 3.50
      },
      b2 : true
   },
   a2: today
}

util.inspect.styles.boolean = 'blue';
// formatação direta com util.inspect
var str = util.inspect(test, {depth: 4, colors: true });
```

```
console.log(str);

// formatação usando console.dir e opções
console.dir(test, {depth: 4, colors: true});

// formatação usando o console.log básico
console.log(test);

// formatação usando o stringify do JSON
console.log(JSON.stringify(test, null, 4));
```

O resultado é mostrado na Figura 4.1 no livro não é possível ver os efeitos de cor. Usamos um fundo branco para a janela de terminal, com texto em preto. Recomendamos executar o exemplo em seu sistema para ver o resultado colorido.

A função `console.dir()` suporta três das quatro opções de `util.inspect()`: `showHidden`, `depth` e `colors`. Ela não suporta `customInspect`. Se ajustado para `true`, a opção indica que o objeto fornece sua própria função de inspeção.

Figura 4.1 – Captura da janela de terminal mostrando as diferentes formatações de strings.

Mensagens mais ricas com o console e um temporizador

Voltando ao objeto `console`, uma técnica bastante útil para mostrar uma

imagem do que está acontecendo na aplicação é adicionar um temporizador e marcar os horários de início e encerramento da execução.

As duas funções do console que usaremos para essa funcionalidade são `console.time()` e `console.timeEnd()`. Ambas devem receber o mesmo nome de temporizador.

No exemplo de código a seguir, o código usa um laço de longa duração para que uma certa quantidade de tempo passe e, assim, possamos registrar a diferença.

```
console.time('the-loop');

for (var i = 0; i < 10000; i++) {
   ;
}

console.timeEnd('the-loop');
```

Mesmo com um laço grande como esse, a diferença de tempo é ínfima e depende da carga da máquina e do processo. Mas a funcionalidade de temporização não está limitada a eventos síncronos. Ser capaz de usar um nome específico permite que usemos a funcionalidade até mesmo em eventos assíncronos.

O código a seguir é uma versão modificada de nosso Hello World do Capítulo 1, iniciando o temporizador no começo da aplicação, encerrando e imediatamente reiniciando a contagem de tempo a cada solicitação web. Assim, temos os intervalos de tempo entre cada solicitação. Obviamente poderíamos usar a função `Date()` para obter um temporizador muito mais completo, mas qual seria a graça disso?

```
var http = require('http');

console.time('hello-timer');
http.createServer(function (request, response) {
   response.writeHead(200, {'Content-Type': 'text/plain'});
   response.end('Hello World\n');
   console.timeEnd('hello-timer');
   console.time('hello-timer');
}).listen(8124);

console.log('Server running at http://127.0.0.1:8124/');
```

Falando sério agora, o que o código demonstra é que podemos incorporar

o temporizador em operações assíncronas, graças à possibilidade de nomear vários deles individualmente.

---

1 N. do T.: A palavra no original é "evaluated", e sua tradução como "avaliada" é imprecisa e ligeiramente incorreta, apesar de toda a literatura nacional, traduzida ou original, usar assim. O verbo "to evaluate" significa "calcular o resultado" ou, mais precisamente, "determinar o valor". Em programação usamos evaluate para obter o valor final de uma expressão, e esse valor nem sempre é numérico. O resultado de uma expressão, depois de "avaliada", pode ser, por exemplo, uma string, um valor booleano (`true` ou `false`) ou mesmo um objeto completo. Decidimos usar a tradução consagrada, mesmo sendo mal traduzida, porque os programadores nacionais já estão acostumados com ela, mas cabe aqui a observação.

2 N. do T.: É possível configurar o PuTTY no Windows para que se comporte melhor (mas ainda aquém) nesse caso. Nas configurações, vá em Window > Appearance, Window > Translation e Connection > Telnet e ajuste por tentativa e erro até obter um comportamento mais consistente.

3 N. do T.: Alguns leitores brasileiros se sentem incomodados com algumas traduções de termos usados no JavaScript. Em especial, os termos view e promise parecem causar bastante estranheza quando traduzidos como visão e promessa, respectivamente. Mantivemos sem tradução os nomes que a comunidade brasileira de desenvolvedores está acostumada a usar em inglês, embora um desenvolvedor norte-americano, quando lê "promise", pense realmente numa promessa, e quando lê "view", pense em visão. Para o programador brasileiro, é mais fácil descolar o significado coloquial da palavra de seu significado na linguagem de programação, e não cabe aqui discutir se essa dissociação é ou não tecnicamente aceitável.

CAPÍTULO 5

Node e a web

O Node ainda não está em posição de substituir a onipresente e matadora combinação Apache/PHP, mas está ganhando popularidade – especialmente se considerarmos a facilidade de criar aplicações multiplataforma e o apoio que tem vindo de grandes multinacionais de tecnologia.

Neste capítulo, vamos explorar as raízes web do Node, incluindo um olhar aprofundado sobre o módulo HTTP, bem como colocar a mão na massa e criar um servidor web estático com recursos simples. Também aprenderemos sobre módulos nativos do Node que simplificam o desenvolvimento para web.

Módulo HTTP: servidor e cliente

Não espere usar o módulo HTTP para recriar um servidor web com a sofisticação de um Apache ou Nginx. Como explicado na documentação do Node, o servidor nativo é de baixo nível e é melhor empregado na manipulação de fluxos de dados e análise de mensagens. Mesmo assim, é a fundação para as funcionalidades mais sofisticadas do framework Express, abordado no Capítulo 10.

O módulo HTTP suporta inúmeros objetos, incluindo `http.Server`, que é devolvido quando usamos a função `http.createServer()` demonstrada no Capítulo 1. Naquele exemplo, embutimos uma função de callback para tratar a solicitação web, mas podemos muito bem empregar eventos separados, pois o `http.Server` herda de `EventEmitter`.

```
var http = require('http');

var server = http.createServer().listen(8124);
```

```
server.on('request', function(request,response) {

   response.writeHead(200, {'Content-Type': 'text/plain'});
   response.end('Hello World\n');
});

console.log('server listening on 8214');
```

Podemos também reconhecer outros eventos, como quando uma conexão é feita (`connect`), ou o cliente solicita uma atualização (`upgrade`). Esta última ocorre quando o cliente pede para usar uma versão mais moderna do HTTP ou para substituí-lo por um protocolo diferente.

As entranhas do HTTP

É sempre bom conhecer um pouco mais sobre os órgãos internos no HTTP. O servidor web do Node, na classe `HTTP.Server`, é na verdade uma implementação baseada no `Net.Server`, que por sua vez implementa comunicação baseada no protocolo TCP. Enquanto o TCP providencia a camada de transporte, o HTTP representa a camada de aplicação. Discutiremos sobre tudo isso no Capítulo 7.

A função de callback usada para responder a uma solicitação web recebe dois parâmetros: `request` e `response` (respectivamente, solicitação e resposta). O segundo parâmetro, `response`, é um objeto do tipo `http.ServerReponse`, e é um *fluxo de escrita* (writable stream) que suporta um grande número de funções, incluindo `response.writeHead()` para criar o cabeçalho da resposta, `response.write()` para escrever os dados da resposta e `response.end()` para marcar o fim dela.

Fluxos de leitura e escrita

O Capítulo 6 destrincha ambos os tipos de fluxo de dados.

O primeiro parâmetro, `request`, é uma instância da classe `IncomingMessage`, que é um *fluxo de leitura* (readable stream). Dentre as informações que podemos acessar a partir da solicitação estão:

`request.headers`
: Os objetos de cabeçalho das solicitações e respostas.

`request.httpVersion`
: A versão do HTTP solicitada.

`request.method`

Válido apenas para uma solicitação `http.Server`, devolve um verbo HTTP (GET ou POST).

`request.rawHeaders`
: Cabeçalhos brutos.

`request.rawTrailers`
: Finalizadores de mensagem (trailers, ou footers) brutos.

Para ver a diferença entre `request.headers` e `request.rawHeaders`, envie-as para o console usando `console.log()` dentro da solicitação. Note que, para `request.headers`, os valores são passados como propriedades, mas para `request.rawHeaders` são passados como um array. A propriedade está no primeiro elemento do array e o valor no segundo, caso queira acessar valores individuais:

```
console.log(request.headers);
console.log(request.rawHeaders);

// chamando o host
console.log(request.headers.host);
console.log(request.rawHeaders[0] + ' is ' + request.rawHeaders[1]);
```

Na documentação do Node, observe que algumas das propriedades de `IncomingMessage` (`statusCode` e `statusMessage`) são acessíveis somente para uma *resposta* (e não uma solicitação) obtida de um objeto `HTTP.ClientRequest`. Além de criar um servidor que atende às solicitações, podemos criar um cliente que *faz* as solicitações. Isso é feito com a classe `ClientRequest`, que instanciamos usando a função `http.request()`.

Para demonstrar ambos os tipos de funcionalidade, servidor e cliente, usaremos um exemplo de código retirado da documentação do Node para criar um cliente e o modificaremos para acessar um servidor que, em relação ao cliente, é local. No cliente, criarei um método POST para enviar dados, o que significa que preciso modificar meu servidor para ler esses dados. É aí que entra o recurso de fluxo de leitura de `IncomingMessage`. Em vez de "escutar" por um evento `request`, a aplicação escuta por um ou mais eventos `data`, que usa para obter *lascas de dados* (em inglês, data chunks; por incrível que pareça, ambos são termos técnicos) na solicitação. A

aplicação continua reunindo essas chunks até que receba um evento `end` no objeto de solicitação. Nesse momento, emprega outro módulo do Node, Query String (que discutiremos mais adiante em "Analisando uma solicitação com Query String"), para analisar os dados e mostrá-los no console. Só depois de tudo isso uma resposta é enviada.

O código do servidor modificado está no Exemplo 5.1. Observe que é muito semelhante ao que já testamos anteriormente, exceto a parte que trata os eventos de dados enviados pelo verbo POST.

Exemplo 5.1 – Um servidor que escuta por um POST e processa os dados postados

```
var http = require('http');
var querystring = require('querystring');

var server = http.createServer().listen(8124);

server.on('request', function(request,response) {

   if (request.method == 'POST') {
      var body = '';

      // adiciona a lasca de dados (data chunk) ao final do corpo da página
      request.on('data', function (data) {
         body += data;
      });

      // transmite os dados
      request.on('end', function () {
         var post = querystring.parse(body);
         console.log(post);
         response.writeHead(200, {'Content-Type': 'text/plain'});
         response.end('Hello World\n');
      });
   }
});
console.log('server listening on 8214');
```

O código do novo cliente está no Exemplo 5.2. Novamente, emprega `http.ClientRequest`, que é uma implementação de fluxos de dados, como fica evidenciado pelo método `req.write()` usado no exemplo.

O código é uma cópia quase idêntica do que encontramos na documentação do Node, exceto pelo servidor acessado. Em nosso caso,

tanto o servidor quanto o cliente estão na mesma máquina, portanto usamos `localhost` como host. Além disso, não especificamos a propriedade `path` nas opções, porque aceitaremos o valor default `/`. Os cabeçalhos para a solicitação estão ajustados para um tipo de conteúdo `application/x-www-form-urlencoded`, usado com dados enviados pelo verbo POST. Observe que o cliente recebe os dados devolvidos pelo servidor via `response`, que é o único argumento da função de callback associada à função `http.request()`. Os dados POSTados são obtidos como "lascas" extraídas do fluxo de resposta, e essas lascas são mostradas no console à medida que chegam. Como a mensagem devolvida é muito curta, apenas um evento `data` é disparado.

A solicitação POSTada não é tratada de forma assíncrona porque estamos iniciando uma ação, não bloqueando o processamento enquanto esperamos pelo término de uma ação.

Exemplo 5.2 – Cliente HTTP enviando dados (POST) a um servidor

```
var http = require('http');
var querystring = require('querystring');

var postData = querystring.stringify({
   'msg' : 'Hello World!'
});

var options = {
   hostname: 'localhost',
   port: 8124,
   method: 'POST',
   headers: {
      'Content-Type': 'application/x-www-form-urlencoded',
      'Content-Length': postData.length
   }
};

var req = http.request(options, function(res) {
   console.log('STATUS: ' + res.statusCode);
   console.log('HEADERS: ' + JSON.stringify(res.headers));
   res.setEncoding('utf8');

   // obtém lascas de dados
   res.on('data', function (chunk) {
      console.log('BODY: ' + chunk);
```

```
    });

    // encerra a resposta
    res.on('end', function() {
        console.log('No more data in response.')
    })
});

req.on('error', function(e) {
    console.log('problem with request: ' + e.message);
});

// grava dados no corpo da solicitação
req.write(postData);
req.end();
```

Abra um segundo terminal, inicie o servidor no primeiro e rode o cliente no segundo. O cliente basicamente diz olá para o servidor, que responde com outro olá.

Esse código todo existe apenas para que dois processos digam olá um para o outro. Contudo, também fomos capazes de implementar uma comunicação de mão dupla entre o cliente e o servidor e tivemos a chance de trabalhar com dados enviados via POST, em vez do costumeiro GET. Como se não bastasse, com exceção de apenas uma, fomos apresentados a todas as classes do módulo HTTP: `http.ClientRequest`, `http.Server`, `http.IncomingMessage` e `http.ServerResponse`.

A única classe que não experimentamos ainda foi `http.Agent`, usada para sockets que fazem consultas periódicas circulares, ou pooling. O Node mantém um banco de sockets de conexão que tratam solicitações iniciadas por `http.request()` ou `http.get()`. Esta última é uma solicitação simplificada, um GET que não tem corpo (body). Se a aplicação estiver disparando um número razoável de solicitações, estas podem ser retidas em um gargalo por conta do tamanho limitado do banco de sockets. Para contornar o problema, é preciso desabilitar o pooling de conexão, configurando `agent` como `false` nas propriedades das solicitações saintes. No Exemplo 5.2, seria preciso alterar as opções a seguir (em negrito):

```
var options = {
    hostname: 'localhost',
    port: 8124,
    method: 'POST',
```

```
    headers: {
      'Content-Type': 'application/x-www-form-urlencoded',
      'Content-Length': postData.length
    },
    agent: false
};
```

Para alterar o tamanho do banco (ou seja, a quantidade de sockets), usamos `agent.maxFreeSockets`. Por default, o valor máximo é `256`. Esteja ciente de que alterar o tamanho do banco de sockets de conexão sainte terá impacto no consumo de memória e de outros recursos da máquina.

Teremos a chance de trabalhar com comunicações mais sofisticadas no Capítulo 7. Por ora, voltemos à criação de um servidor HTTP que devolva algo mais que "e aí, tudo bem?".

O que é necessário para criar um servidor web estático?

O Node oferece toda a funcionalidade necessária para construir um roteador simples ou para servir páginas estáticas. Contudo, ser *possível* fazer é muito diferente de ser *fácil* fazer.

Se fizermos uma lista do que é necessário para montar um servidor web estático simples, mas funcional, chegaríamos aos seguintes passos:

1. Criar um servidor HTTP que fique de prontidão para receber solicitações dos clientes.
2. Quando uma solicitação chegar, o URL deve ser analisado para determinar a localização do arquivo.
3. Verificar se o arquivo solicitado existe mesmo.
4. Se o arquivo não existir, responder de acordo.
5. Se o arquivo realmente existir, abri-lo para leitura.
6. Preparar um cabeçalho de resposta.
7. Transferir o conteúdo do arquivo para o corpo da resposta.
8. Esperar por uma nova solicitação.

Só precisamos de módulos nativos para implementar todas essas funções (com uma única exceção, que veremos mais adiante). Criar um servidor

HTTP e ler arquivos do disco são duas ações que requerem dois módulos bastante populares: o módulo HTTP e o módulo File System. Além deles, queremos definir uma variável global para o diretório-base ou, alternativamente, usar a variável predefinida `__dirname` (discutiremos sobre ela mais adiante, em "Por que não usar `__dirname`?").

Neste momento, o início da aplicação tem o código mostrado a seguir:

```
var http = require('http'),
   fs = require('fs'),
   base = '/home/examples/public_html';
```

No Capítulo 1, criamos um servidor web para nossa versão de Hello World, e usaremos aquele código como base para todo o trabalho a ser desenvolvido aqui. A função `http.createServer()` cria o servidor, com um callback que recebe dois valores: a solicitação web e a resposta que criaremos. A aplicação obtém o documento solicitado acessando diretamente a propriedade `url` do objeto de solicitação HTTP. Para verificar a resposta, comparando-a com a solicitação, geraremos um `console.log` com o caminho completo do arquivo solicitado, que aparecerá logo abaixo da mensagem de `console.log` exibida no momento em que o servidor é iniciado:

```
var http = require('http'),
   fs = require('fs'),
   base = '/home/examples/public_html';

http.createServer(function (req, res) {

   pathname = base + req.url;
   console.log(pathname);

}).listen(8124);

console.log('Server web running at 8124');
```

Rodar a aplicação inicia um servidor web que recebe solicitações web na porta 8124.

Sobre as portas que usamos

Quando acessamos um site usando o navegador, não precisamos quase nunca informar o número de porta. Isso ocorre porque há duas *portas padronizadas* – 80 para HTTP e 443 para HTTPS – que os navegadores assumem automaticamente a cada solicitação. A Wikipédia mantém uma lista com essas portas padronizadas.

Em nosso servidor, estamos usando a porta 8214. Fazemos isso porque a porta 80 pode já estar sendo usada, na mesma máquina, pelo servidor web primário de seu ambiente de desenvolvimento – que pode ser Ngnix, Apache, lighttpd, o Caddy, que é bem recente, e até mesmo uma aplicação Node já em produção.

Poderíamos muito bem usar a porta-padrão 80, mas as portas abaixo de 1024 requerem permissões de root (ou seja, do superusuário administrador da máquina). Nem mesmo o Apache cria instâncias que rodam como root. O próprio Apache roda como root, mas cria threads-filhas que rodam em níveis de permissão bem mais restritos. Por questões de segurança, ninguém realmente executa servidores web com permissões de root.

Poderíamos usar os recursos internos do próprio sistema operacional do servidor, como o iptables no caso do Linux, para redirecionar para a porta 80 qualquer acesso à porta 1024, mas pessoalmente não tomaria esse caminho. A melhor maneira é, ainda, configurar um serviço web principal (Apache, Ngnix, lighttpd ou Caddy) e fazer com que ele seja um proxy para as solicitações vindas do cliente, redirecionando-as para a aplicação em Node. Veremos como configurar o Apache para agir como proxy de aplicações do Node mais adiante, em "O Apache como proxy de uma aplicação Node".

Se testarmos a aplicação com um arquivo *index.html* (por exemplo, *http://blipdebit.com:8124/index.html*), obteremos no console a seguinte mensagem (que varia conforme seu ambiente):

```
/home/examples/public_html/index.html
```

Obviamente, o navegador fica esperando eternamente por uma página que nunca virá, pois ainda não preparamos a resposta. Faremos isso a seguir.

Podemos testar se o arquivo solicitado está disponível antes de abri-lo. A função `fs.stat()` devolve um erro se o arquivo não existir.

E se o arquivo for removido depois da verificação?

Um dos problemas de usar `fs.stat()` é que pode acontecer de, por qualquer motivo, o arquivo sumir entre a verificação de sua existência e a abertura de fato do arquivo. Outra maneira seria abrir diretamente o arquivo; se ele não existir, um erro é gerado. O problema é que, ao usar `fs.open()` diretamente e depois passar o descritor de arquivos gerado para `fs.createReadStream()`, não há como saber se o objeto sendo aberto é um arquivo mesmo ou uma pasta, muito menos se ele existe ou não ou se está bloqueado. O que costumo fazer é uma verificação dupla: usar `fs.stat()`, mas também verificar se um erro é gerado ao tentar ler o arquivo. Dessa forma, tenho mais controle sobre o tratamento de erros.

Ainda sobre a leitura de arquivos, poderíamos tentar a função `fs.readFile()` para ler o conteúdo do arquivo. Entretanto, o problema com

`fs.readFile()` é que ele lê o arquivo por completo e o guarda na memória antes de disponibilizar os dados. Contudo, os documentos na web costumam ser grandes. Além disso, podem haver muitas solicitações simultâneas para um mesmo documento. Por tudo isso, `fs.readFile()` pode não ser o método mais apropriado.

Em vez de `fs.readFile()`, a aplicação cria um fluxo de leitura com o método `fs.createReadStream()`, usando as configurações-padrão. Para isso, basta redirecionar (com *pipe*) o conteúdo do arquivo diretamente para o objeto de resposta HTTP. Como o fluxo manda um sinal de encerramento quando não há mais dados a transmitir, não precisamos chamar o método `end`:

```
res.setHeader('Content-Type', 'test/html');

// cria um fluxo de leitura e o redireciona
var file = fs.createReadStream(pathname);
file.on("open", function() {
   // Status 200 - arquivo encontrado, sem erros
   res.statusCode = 200;
   file.pipe(res);
});
file.on("error", function(err) {
   res.writeHead(403);
   res.write('file missing ou permission problem');
   console.log(err);
});
```

O fluxo de leitura tem dois eventos de interesse: `open` e `error`. O evento `open` é emitido quando o fluxo está pronto para leitura, e `error` é emitido quando um erro acontece. Que tipo de erro? O arquivo pode, por exemplo, ter sumido após a verificação de estado, as permissões podem impedir o acesso, ou aquele nome pertence a uma pasta, não um arquivo. Como não sabemos o que pode acontecer até esse momento, na ocorrência de qualquer erro geramos um status `403`, que é genérico o suficiente para abranger a maioria dos problemas potenciais. Escrevemos uma mensagem que pode ou não ser mostrada, dependendo de como o navegador responde ao erro.

A aplicação chama o método `pipe` na função de callback para o evento

`open`.

Neste momento, o servidor web estático se parece com o código do Exemplo 5.3.

Exemplo 5.3 – Um servidor web estático muito simples

```
var http = require('http'),
   fs = require('fs'),
   base = '/home/examples/public_html';

http.createServer(function (req, res) {

   pathname = base + req.url;
   console.log(pathname);

   fs.stat(pathname, function(err,stats) {
      if (err) {
         console.log(err);
         res.writeHead(404);
         res.write('Resource missing 404\n');
         res.end();
      } else {
         res.setHeader('Content-Type', 'text/html');

         // cria um fluxo de leitura e o redireciona
         var file = fs.createReadStream(pathname);

         file.on("open", function() {
            res.statusCode = 200;
            file.pipe(res);
         });

         file.on("error", function(err) {
            console.log(err);
            res.writeHead(403);
            res.write('file missing or permission problem');
            res.end();
         });
      }
   });
}).listen(8124);

console.log('Server running at 8124');
```

Com um arquivo HTML simples para teste, que tem um simples elemento `img`, o arquivo é carregado e mostrado corretamente:

```
<!DOCTYPE html>
```

```
<head>
   <title>Test</title>
   <meta charset="utf-8" />
</head>
<body>
<img src="./phoenix5a.png" />
</body>
```

Também experimentei com um arquivo que não existia. O erro é capturado na chamada a `fs.stat()`, devolvendo uma mensagem `404` ao navegador e mostrando um erro no console.

Depois, tentei transferir um arquivo em que todas as permissões haviam sido removidas. Desta vez, o fluxo de leitura capturou o erro, enviando para o navegador uma mensagem afirmando não ser possível ler o arquivo, e ao console, uma descrição mais elaborada a respeito das permissões. Nesse caso, o status devolvido é bem mais apropriado: `403`, que corretamente engloba problemas de permissão.

Por fim, tentei a aplicação de servidor com outro arquivo de exemplo que eu tinha, contendo um elemento `video`, novidade do HTML5:

```
<!DOCTYPE html>
<head>
   <title>Video</title>
   <meta charset="utf-8" />
</head>
<body>
   <video id="meadow" controls>
      <source src="videofile.mp4" />
      <source src="videofile.ogv" />
      <source src="videofile.webm" />
   </video>
</body>
```

Embora o arquivo abra e o vídeo seja mostrado quando usei o navegador Chrome, o elemento `video` não funcionou no Internet Explorer 10. Olhando o console, descobrimos a razão:

```
Server running at 8124/
/home/examples/public_html/html5media/chapter1/example2.html
/home/examples/public_html/html5media/chapter1/videofile.mp4
/home/examples/public_html/html5media/chapter1/videofile.ogv
```

```
/home/examples/public_html/html5media/chapter1/videofile.webm
```

Embora o IE10 seja capaz de reproduzir arquivos MP4, ele testa todos os três tipos de vídeo porque o content type (tipo de conteúdo) do cabeçalho de resposta é `text/html` para todos eles. Outros navegadores simplesmente ignoram o tipo incorreto e mostram a mídia indicada, mas o IE10 não – e esse é o certo, na minha opinião, porque eu não teria encontrado esse erro na aplicação tão facilmente se não fosse pelo IE10.

A Microsoft atualizou a rotina de tratamento de conteúdo em seu novo navegador, o Edge, e este mostra o vídeo corretamente. Ainda assim, queremos que a aplicação faça a coisa certa. Precisamos modificá-la para que teste a extensão de todos os arquivos e devolva o tipo MIME apropriado no cabeçalho de resposta. Poderíamos criar nosso próprio código para essa funcionalidade, mas já existe um módulo pronto que faz isso com facilidade: Mime.

O módulo Mime pode ser instalado usando o npm: `npm install mime`. O módulo está hospedado no GitHub.

O módulo Mime pode devolver o tipo MIME apropriado de acordo com o nome do arquivo (com ou sem o caminho) e pode, ainda, devolver a extensão do arquivo se informarmos o tipo de conteúdo. O módulo Mime é importado em nosso código com `require`:

```
var mime = require('mime');
```

O tipo de conteúdo devolvido é usado no cabeçalho de resposta e também enviado ao console, portanto podemos verificar o valor quando testamos a aplicação:

```
// tipo de conteúdo
var type = mime.lookup(pathname);
console.log(type);
res.setHeader('Content-Type', type);
```

Agora, quando acessamos o arquivo com o elemento vídeo no IE10, o vídeo funciona. Em todos os navegadores, o tipo correto de conteúdo é devolvido.

Entretanto, acessar uma pasta em vez de um arquivo ainda não funciona. Quando isso acontece, um erro é mostrado no console:

```
{ [Error: EISDIR, illegal operation on a directory] errno: 28, code: 'EISDIR' }
```

O erro diz: “EISDIR, operação ilegal em uma pasta”. Contudo, o que acontece no navegador varia. O Edge mostra um erro genérico 403 gerado por ele mesmo, como mostrado na Figura 5.1, enquanto o Firefox e o Chrome mostram a mensagem vinda da aplicação, “something is wrong” (algo está errado). Mas existe uma diferença marcante entre não conseguir acessar uma pasta e ter algo errado com um arquivo.
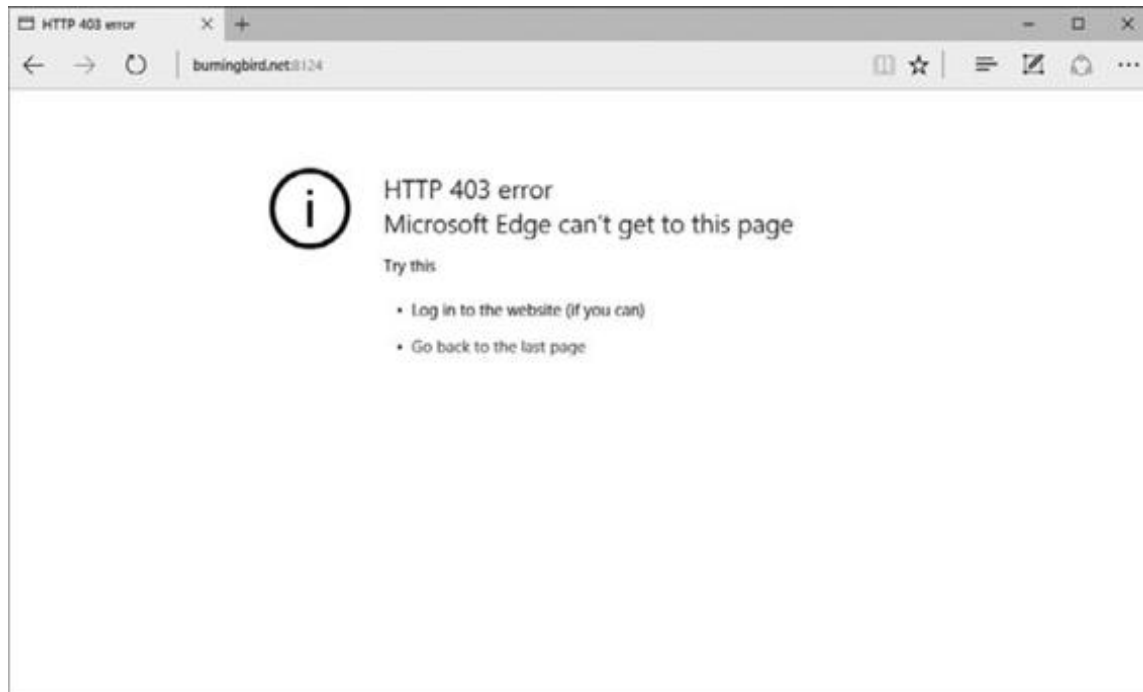
Figura 5.1 – Como o Edge trata o acesso não permitido a uma pasta.

Precisamos não só verificar se o recurso sendo acessado existe, mas também determinar se é uma pasta ou um arquivo. Se for uma pasta, podemos entrar nela e mostrar seu conteúdo, ou devolver um erro – a escolha é do desenvolvedor.

É preciso fazer ainda mais uma alteração. O caminho-base que estamos usando funciona perfeitamente em meu servidor Linux, mas teremos problemas se tentarmos o mesmo caminho em uma máquina Windows. Para começar, não existe */home/examples/public_html* no Windows e, em segundo lugar, o Windows usa a barra invertida (\) para indicar o caminho.

Para que a aplicação funcione perfeitamente em ambos os ambientes, é

preciso criar a estrutura de pastas no Windows. Depois, uso o módulo nativo Path para *normalizar* a string de caminho e, assim, funcionar em ambos os ambientes. A função `path.normalize()` recebe uma string e retorna a mesma string, se já estiver normalizada para aquele ambiente, ou uma string transformada.

```
var pathname = path.normalize(base + req.url);
```

Agora a aplicação funciona em todos os ambientes.

O módulo Path será discutido no Capítulo 6.

A versão final de nosso servidor estático está no Exemplo 5.4. Ela usa `fs.stats` para verificar a existência do objeto solicitado e testar se é um arquivo. Se o recurso não existir, o erro HTTP devolvido tem status `404`. Se o recurso existir, mas for uma pasta, o erro HTTP terá status `403` – acesso proibido. O mesmo ocorre se o arquivo estiver bloqueado, mas a mensagem nesse caso será outra. Em todos os casos, a resposta à solicitação será bem mais apropriada à situação.

Exemplo 5.4 – Versão final de nosso servidor web mínimo

```
var http = require('http'),
   url = require('url'),
   fs = require('fs'),
   mime = require('mime'),
   path = require('path');

var base = '/home/examples/public_html';

http.createServer(function (req, res) {

   pathname = path.normalize(base + req.url);
   console.log(pathname);

   fs.stat(pathname, function(err, stats) {
      if (err) {
         res.writeHead(404);
         res.write('Resource missing 404\n');
         res.end();
      } else if (stats.isFile()) {
         // tipo de conteúdo
         var type = mime.lookup(pathname);
         console.log(type);
```

```
            res.setHeader('Content-Type', type);

            // cria e redireciona o fluxo de leitura
            var file = fs.createReadStream(pathname);
            file.on("open", function() {
               // status 200 - arquivo encontrado, sem erros
               res.statusCode = 200;
               file.pipe(res);
            });
            file.on("error", function(err) {
               console.log(err);
               res.statusCode = 403;
               res.write('file permission');
               res.end();
            });
         } else {
            res.writeHead(403);
            res.write('Directory access is forbidden');
            res.end();
         }
      });
   }).listen(8124);
   console.log('Server running at 8124');
```

No console, as mensagens a seguir são mostradas quando acessamos uma página web que contém uma imagem e um vídeo, usando o Firefox:

```
/home/examples/public_html/video.html
text/html
/home/examples/public_html/phoenix5a.png
image/png
/home/examples/public_html/videofile.mp4
video/mp4
/home/examples/public_html/favicon.ico
image/x-icon
/home/examples/public_html/favicon.ico
image/x-icon
```

Observe que todos os tipos de conteúdo foram tratados corretamente – sem contar o acesso automático ao *favicon*, que tanto Firefox quanto Chrome incluem em todas as solicitações de página.

Temos uma noção bem mais ampla de como o fluxo de leitura funciona quando carregamos uma página que tem um elemento HTML de vídeo e

este começa a ser reproduzido. O navegador se conecta ao fluxo de leitura na máxima velocidade que conseguir, preenchendo seu buffer interno, e então pausa a saída. Se o servidor for encerrado enquanto o vídeo estiver tocando, este continua sem ser interrompido até que o buffer seja esvaziado. O elemento de vídeo escurece porque o fluxo de leitura não está mais disponível. Na verdade, é fascinante perceber como tudo funciona direitinho com tão pouco esforço de nossa parte.

Por que não usar __dirname?

Em alguns dos exemplos deste livro, o caminho em que o documento web se encontra está fixo no código. Tipicamente, usamos */home/examples/public_html*. Você pode estar se perguntando por que não usei `__dirname`.

Em Node, podemos usar a variável predefinida `__dirname` para especificar a pasta atual em uma aplicação Node. Entretanto, nos exemplos deste capítulo, os arquivos acessados estão em um caminho completamente separado em relação à aplicação, e é por isso que `__dirname` não foi usado. Exceto nessa circunstância, `__dirname` deve sempre ser usado. Com ele, é possível testar aplicações e implementá-las em produção sem ter de alterar o valor da variável que armazena o caminho-base.

Use `__dirname` da seguinte maneira:

```
var pathname = __dirname + req.url;
```

Observe que o nome da variável `__dirname` começa com dois underscores.

Embora a aplicação funcione quando testada com diferentes documentos, está longe de ser perfeita. Ela não consegue tratar outros tipos de solicitação web, não sabe nada sobre segurança ou cache e não trata corretamente as solicitações de vídeo. Uma aplicação que testei, que rodava em uma única página e usava o elemento vídeo do HTML, também empregava a API de vídeo do HTML para obter o estado do processo de carga do vídeo. Essa aplicação não obteve a informação de que precisava para funcionar.

O número de pegadinhas que podem nos derrubar quando criamos um servidor estático é enorme, e é por isso que muitas pessoas preferem usar frameworks mais complexos como o Express, que veremos no Capítulo 9.

O Apache como proxy de uma aplicação Node

Ninguém em sã consciência diria que o Node é um substituto para os servidores web já estabelecidos, como o Apache. Mas fazer o Node lidar com funcionalidades específicas enquanto o servidor principal lida com o resto é uma opção viável. Como o Apache ainda é o servidor mais popular, nos concentraremos nele por enquanto.

Como podemos rodar um servidor em Node ao mesmo tempo que o Apache e sem forçar o usuário a usar uma porta específica? Apenas um servidor web pode responder na porta 80, que é a padrão para o serviço HTTP. É verdade que podemos disponibilizar uma API para os navegadores e, com ela, esconder o número de porta, mas e se quisermos usar o Apache e o Node simultaneamente sem conflitos?

Outros problemas nos ambientes de produção

Manter o Node rodando permanentemente e fazê-lo se recuperar sozinho de falhas é uma arte discutida no Capítulo 11.

O meio mais simples de rodar uma aplicação Node lado a lado com o Apache é fazer com que este aja como um *proxy* para as solicitações destinadas à aplicação. Todas as solicitações cujo destino é a aplicação no Node devem, necessariamente, passar primeiro por dentro do Apache.

A solução é boa e ruim ao mesmo tempo. A parte boa é o fato de ser um arranjo extremamente simples, e temos um servidor web conhecido e bastante robusto para "matar no peito" todas as solicitações antes de repassá-las à aplicação em Node. O Apache torna o ambiente mais seguro e tem funcionalidades diversas que seriam extremamente difíceis de implementar em uma aplicação Node. O lado ruim é que o Apache cria uma nova thread para cada solicitação que recebe, e os recursos da máquina são finitos e limitados.

Ainda assim, a maioria dos sites da internet estão hospedados em um Apache e conseguem sobreviver sem lentidões constantes. A não ser que a audiência de seu sistema seja contada aos milhões, esta é uma ótima opção.

Para tornar o Apache um proxy do Node, é preciso, primeiro, habilitar o recurso de proxy no Apache. Os comandos a seguir, no terminal, ativam o recurso:

```
a2enmod proxy
a2enmod proxy_http
```

Depois, adicione o recurso ao subdomínio. Por exemplo, em meu servidor, defini o endereço shelleystoybox.com como o local de hospedagem de minha aplicação:

```
<VirtualHost ipaddress:80>
   ServerAdmin shelleyp@burningbird.net
   ServerName shelleystoybox.com

   ErrorLog path-to-logs/error.log
   CustomLog path-to-logs/access.log combined

   ProxyRequests off

   <Location />
      ProxyPass http://ipaddress:2368/
      ProxyPassReverse http://ipaddress:2368/
   </Location>
</VirtualHost>
```

Altere o subdomínio, o email do administrador, a porta e o endereço IP para refletir o seu ambiente. Depois, basta recarregar o subdomínio:

```
a2ensite shelleystoybox.com
service apache2 reload
```

Claro, o subdomínio mostrado no comando deve refletir o seu ambiente, não o meu.

Acessando o site pela porta original

O uso do proxy não impede que as pessoas acessem o site explicitamente pela porta original. Para bloquear esse acesso, é preciso um pouco de conhecimento avançado a respeito de seu ambiente. Por exemplo, em meu servidor Ubuntu, eu poderia criar uma regra no iptables:

```
iptables -A input -i eth0 -p tcp --dport 2368 -j DROP
```

Só que, para isso, eu precisaria ter conhecimentos avançados de administração de sistema.

Analisando uma solicitação com Query String

Já tivemos a oportunidade de usar o módulo Query String neste mesmo capítulo. Seu único propósito é preparar e processar strings de query, ou seja, dados enviados no próprio URL após um caractere `'?'`.

Quando recebemos uma string de query, podemos convertê-la em um

objeto usando querystring.parse(), como demostrado no Exemplo 5.1. O separador default de strings (`'&'`) pode ser substituído em um segundo parâmetro opcional da função, e o operador de atribuição default ('=') pode ser também substituído no terceiro. O quarto parâmetro opcional contém um decodeURIComponent, que por default é querystring.unescape(). Outro deve ser escolhido se a string não estiver codificada em UTF-8. Entretanto, o mais comum é que as strings de query usem sempre separador e atribuição default, e serão sempre em UTF-8, portanto podemos deixar os parâmetros opcionais de fora.

Para entender como a função querystring.parse() faz seu trabalho, considere a seguinte string de query:

```
somedomain.com/?value1=valueone&value1=valueoneb&value2=valuetwo
```

A função querystring.parse() analisa a string de query e devolve o seguinte objeto JSON:

```
{
   value1: ['valueone','valueoneb'],
   value2: 'valuetwo'
}
```

Quando preparamos uma string de query para envio, como demonstrado no Exemplo 5.2, devemos usar querystring.stringify(), passando o objeto a ser codificado. Se tivermos um objeto como o que acabamos de analisar, basta repassá-lo a querystring.stringify() e obteremos uma string de query corretamente formatada, pronta para ser transmitida em um URL. No Exemplo 5.2, querystring.stringify() devolve:

```
msg=Hello%20World!
```

Observe o código de escape do caractere espaço. A função querystring.stringify() recebe os mesmos parâmetros opcionais, com a exceção do objeto da última opção. Nesse caso, poderíamos informar um encodeURIComponent personalizado, que é stringify.escape() por default.

DNS

É muito difícil que sua aplicação precise de um serviço de DNS integrado, mas, se precisar, toda a funcionalidade necessária pode ser obtida do

módulo nativo DNS.

Veremos duas de suas funções mais importantes: `dns.lookup()` e `dns.resolve()`. A função `dns.lookup()` pode ser usada para determinar o primeiro endereço IP de um dado domínio. No código a seguir, o exemplo retorna o primeiro endereço IP do domínio oreilly.com:

```
dns.lookup('oreilly.com', function(err,address,family) {
   if (err) return console.log(err);

   console.log(address);
   console.log(family);
});
```

A variável `address` recebe o endereço IP devolvido, e o valor em `family` pode ser 4 ou 6, a depender de o endereço ser IPv4 ou IPv6. Podemos também especificar um objeto `options`:

`family`
: Um número, 4 ou 6, que representa a versão de endereçamento IP desejada (IPv4 ou IPv6).

`hints`
: Um número; indica as flags `getaddrinfo` suportadas.

`all`
: Se for `true`, devolve todos os endereços (o default é `false`).

Estava curiosa sobre a opção `all`, portanto modifiquei o código para obter todos os endereços. Contudo, ao fazer isso eliminei o parâmetro `family`, que é `undefined` quando acessamos todos os endereços. Obtive um array com todos os objetos de endereço IP aplicáveis, cada um mostrando o endereço IP e a família (versão):

```
dns.lookup('oreilly.com', {all: true}, function(err,family) {
   if (err) return console.log(err);

   console.log(family);
});
```

Esse código produz o seguinte resultado:

```
[ { address: '209.204.146.71', family: 4 },
  { address: '209.204.146.70', family: 4 } ]
```

A função `dns.resolve()` resolve um hostname, obtendo os tipos de registro. Os tipos (devolvidos como strings) podem ser:

`A`
: Endereço IPv4 default

`AAAA`
: Endereço IPv6

`MX`
: Endereço do servidor principal de email (mail exchange)

`TXT`
: Registros de texto

`SRV`
: Registro de servidores (SRV)

`PTR`
: Usado para pesquisa de IP reverso

`NS`
: Servidor de nomes

`CNAME`
: Registro de nome canônico

`SOA`
: Registro de Início de Autoridade (Start Of Authority)

No exemplo a seguir, usamos `dns.resolve()` para obter todos os registros de MX do domínio oreilly.com:

```
dns.resolve('oreilly.com','MX', function(err,addresses) {
   if (err) return err;
   console.log(addresses);
});
```

Ao ser executado, obtemos:

```
[ { exchange: 'aspmx.l.google.com', priority: 1 },
```

```
{ exchange: 'alt1.aspmx.l.google.com', priority: 5 },
{ exchange: 'aspmx2.googlemail.com', priority: 10 },
{ exchange: 'alt2.aspmx.l.google.com', priority: 5 },
{ exchange: 'aspmx3.googlemail.com', priority: 10 } ]
```

CAPÍTULO 6

Node e o sistema de arquivos local

O módulo File System do Node foi usado por todo o livro. Poucos recursos são tão essenciais para uma aplicação quanto o acesso ao sistema de arquivos; a única exceção talvez seja o acesso à rede, abordado nos Capítulos 5 e 7.

Uma das coisas mais sensacionais do Node é que o módulo File System, na maioria das vezes, funciona da mesma forma nos diferentes sistemas operacionais. O Node também se esforça para garantir, sempre que possível, que todas as suas funcionalidades sejam agnósticas em relação ao sistema operacional. Às vezes ele consegue, outras vezes precisamos da ajuda de módulos de terceiros.

Este capítulo é uma introdução formal ao módulo File System. Além da funcionalidade do módulo, veremos as particularidades de cada sistema operacional. Por fim, estudaremos dois módulos, ReadLine e ZLib, que oferecem, respectivamente, interatividade na linha de comando e recursos para comprimir arquivos.

Explorando o sistema operacional

Algumas tecnologias conseguem abstrair todos os detalhes, mesmo os minúsculos, do sistema operacional. Outras precisam de muito trabalho para acomodar as especificidades de cada sistema. O Node está em algum lugar no meio do caminho. Na maioria dos casos, basta criarmos uma aplicação e ela rodará em qualquer lugar. Entretanto, existem algumas funcionalidades nas quais as diferenças entre sistemas operacionais acabam aparecendo. Como mencionado no início do capítulo, às vezes o Node lida muito bem com elas, mas eventualmente precisaremos da ajuda

de módulos de terceiros.

Acessar diretamente informações sobre o sistema operacional é simples, se usarmos o módulo nativo OS. É um dos mais úteis para a criação de aplicações multiplataforma, além de ser uma mão na roda para obter informações sobre o uso de recursos e a capacidade do ambiente atual.

Para acessar o módulo OS, precisamos de um `require`:

```
var os = require('os');
```

O módulo OS é apenas informativo. Por exemplo, se quisermos garantir compatibilidade entre plataformas, podemos descobrir qual o caractere de fim de linha suportado pelo sistema, ou se é big-endian (BE)/little-endian (LE), e até mesmo ter acesso à pasta de arquivos temporários e à pasta pessoal dos usuários:

```
var os = require('os');

console.log('Using end of line' + os.EOL + 'to insert a new line');
console.log(os.endianness());
console.log(os.tmpdir());
console.log(os.homedir());
```

Tanto meu servidor Ubuntu quanto minha máquina com Windows 10 são LE, e o caractere de EOL funciona em ambos como esperado (a segunda parte da linha começa com um caractere new line). Naturalmente, a única diferença é a localização das pastas temporárias e de usuários.

A pasta temporária

A pasta temporária é onde os arquivos em uso são armazenados temporariamente. O conteúdo dessa pasta é apagado sempre que o sistema é reiniciado, ou a intervalos regulares.

O módulo OS também permite verificar os recursos disponíveis na máquina.

```
console.log(os.freemem());
console.log(os.loadavg());
console.log(os.totalmem());
```

A função `os.loadavg()` é específica do Unix; no Windows, ela devolve zeros. Seu propósito é informar a carga média de trabalho, ou atividade, do

sistema em intervalos de 1, 5 e 15 minutos. Para obter uma porcentagem, basta multiplicar os três números por 100. As funções de memória `os.freemen()` e `os.totalmem()` devolvem, respectivamente, a quantidade de memória livre (sem uso) e a quantidade total de memória do sistema, ambas em bytes.

Outra função, `os.cpus()`, devolve informação sobre os processadores da máquina. Mais especificamente, devolve o número de milissegundos que a CPU empregou nos processos de `user`, `nice`, `sys`, `idle` e `irq`. Se não estiver familiarizado com os conceitos, o valor `user` é a quantidade de tempo gasto pela CPU em processos dos usuários, `idle` é a quantidade de tempo em que a CPU não fez nada (conhecido no Windows como "tempo ocioso do sistema") e `sys` é o tempo gasto pela CPU com processos do núcleo (kernel) do sistema operacional. O valor de `nice` reflete a porcentagem de processos dos usuários que tiveram sua prioridade alterada com o comando `nice` do Unix – as prioridades podem ser ajustadas manualmente pelo administrador do sistema para que um processo rode com menos frequência, deixando mais tempo de CPU para outros processos importantes. O `irq` é um sinal de interrupção para a CPU que solicita o desvio do processamento para acessar um serviço disponível no próprio hardware.

Os tempos de CPU não são tão úteis, melhor seria obter uma porcentagem. Podemos calculá-las somando todos os valores de tempo e dividindo o valor desejado por essa soma. Há módulos de terceiros que já calculam essas porcentagens, bem como outras informações. A Microsoft disponibilizou um documento bem informativo sobre como usar essas funções no Azure (*https://blogs.msdn.microsoft.com/azureossds/2015/08/23/finding-memory-leaks-and-cpu-usage-in-azure-node-js-web-app/*), mas as instruções e os módulos listados devem funcionar em qualquer ambiente, não apenas no Azure.

Fluxos e redirecionamentos

Os fluxos (streams) aparecem em todos os módulos nativos do Node,

oferecendo, por exemplo, funcionalidade de HTTP e outras formas de comunicação em rede. Os fluxos também são usados em funcionalidades de sistema de arquivos, e é por isso que vamos estudá-lo agora, antes de mergulharmos mais fundo no módulo File System.

O objeto Stream é uma interface abstrata, o que significa que não vamos criar fluxos diretamente. Em vez disso, trabalharemos com diversos objetos que implementam Stream, como, por exemplo, uma solicitação HTTP, fluxos de leitura e escrita em arquivos pelo módulo File System, compactação de arquivos pelo módulo ZLib e até mesmo `process.stdout`. A implementação direta pela API do módulo Stream só se justifica quando queremos criar um fluxo personalizado. Como isso está além do escopo deste texto introdutório, deixarei o exercício de implementar um fluxo pela API de Stream como desafio para o leitor. Por ora, vamos nos concentrar em como se comportam os fluxos expostos em outras funcionalidades.

Como há um número grande de objetos do Node que implementam uma interface de fluxo, há algumas funcionalidades básicas que são comuns a todos os streams:

- Podemos alterar a codificação do fluxo com `setEncoding`.
- Podemos verificar se o fluxo é de leitura, de escrita ou ambos.
- Podemos capturar eventos de fluxo, como *data received* (dados recebidos) ou *connection closed* (conexão encerrada), associando funções de callback a cada evento.
- Podemos pausar e retomar o fluxo.
- Podemos redirecionar dados de um fluxo de leitura para um fluxo de escrita, num comportamento semelhante ao pipe (|) do Unix.

Preste atenção no item a respeito dos fluxos poderem ser de leitura, de escrita ou ambos. O último tipo de stream é conhecido como *fluxo duplex*. Há ainda uma variação do fluxo duplex conhecida como `transform` (*fluxo de transformação ou, simplesmente, transformada*), quando a entrada e a saída têm relação causal. Veremos esse tipo de stream quando estudarmos o módulo ZLib de compactação, ainda neste capítulo.

Um fluxo de leitura começa em pausa, o que significa que nenhum dado é enviado até que alguma forma de leitura explícita (`stream.read()`) ou um comando para retomar o fluxo (`stream.resume()`) ocorra. Entretanto, as implementações que usamos aqui, como o fluxo de leitura de File System, são liberadas (ou seja, entram em modo de fluxo) assim que o código pede um evento de dados (a maneira de acessar os dados em um fluxo de leitura). Em modo de fluxo, os dados são acessados e enviados à aplicação no mesmo momento em que ficam disponíveis.

O fluxo de leitura suporta inúmeros elementos, mas na maioria dos nossos exemplos há três eventos de maior interesse: `data`, `end` e `error`. O evento `data` é enviado quando há uma lasca (chunk) de dados pronta para ser utilizada, enquanto `end` é enviado quando todos os dados da fila foram consumidos. O evento `error` é enviado quando surge um erro, como era de esperar. Já testemunhamos um fluxo de leitura em ação no Exemplo 5.1 do Capítulo 5, e o veremos novamente mais tarde, quando tratarmos do módulo File System.

Um fluxo de escrita é o destino para o qual os dados são enviados ("escritos"). Dentre os eventos que podemos detectar estão `error` e `finish`, quando o método `end()` é chamado e todos os dados foram enviados. Outro evento possível é `drain`, quando uma tentativa de escrita de dados devolve o valor `false`. Usamos um fluxo de escrita quando criamos um cliente HTTP no Exemplo 5.2 do Capítulo 5, e também o veremos em ação mais adiante, quando trabalharmos com o módulo File System em "Uma introdução formal ao módulo File System".

Um fluxo duplex tem elementos dos tipos de fluxo de escrita e de leitura. Um transform é um tipo de fluxo duplex no qual os buffers internos de entrada e saída são conectados entre si – ao contrário dos fluxos duplex "normais", nos quais os buffers internos de entrada e saída são independentes um do outro. Num transform, entre os fluxos de entrada e saída há um passo adicional que executa a transformação dos dados propriamente dita. Internamente, os fluxos de transformação devem implementar uma função `_transform()`, que recebe um dado de entrada, faz algo com ele e o empurra para a saída.

Para entender melhor os transforms, precisamos mergulhar em uma funcionalidade suportada por todos os streams: a função `pipe()`. Já a vimos em ação no Exemplo 5.1, quando redirecionamos com pipe o conteúdo de um fluxo de leitura diretamente para um objeto de resposta HTTP:

```
// cria um fluxo de leitura e o redireciona
var file = fs.createReadStream(pathname);
file.on("open", function() {
   file.pipe(res);
});
```

O pipe extrai os dados do arquivo (neste caso, um fluxo) e os redireciona para o objeto `http.ServerResponse`. Na documentação do Node, aprendemos que esse objeto implementa a interface de fluxo de escrita. Mais adiante, veremos que `fs.createReadStream()` devolve um `fs.ReadStream`, que é uma implementação de um fluxo de leitura. Um dos métodos que todo fluxo de leitura suporta é justamente um `pipe()` para um fluxo de escrita.

Mais tarde, veremos um exemplo de uso do módulo ZLib para comprimir um arquivo, mas, por enquanto, eis um exemplo simples. Esta é uma excelente demonstração de um fluxo de transformação.

```
var gzip = zlib.createGzip();
var fs = require('fs');
var inp = fs.createReadStream('input.txt');
var out = fs.createWriteStream('input.txt.gz');

inp.pipe(gzip).pipe(out);
```

A entrada é um fluxo de leitura e a saída é um fluxo de escrita. O conteúdo de um é encaminhado ao outro, mas no meio do caminho os dados passam por uma rotina de compactação. Isso, simplificadamente, é um fluxo de transformação.

Uma introdução formal ao módulo File System

O módulo File System (fs) do Node oferece toda a funcionalidade que um desenvolvedor pode querer para interagir com o sistema de arquivos de uma máquina, independentemente do sistema operacional. Já vimos o fs em ação ao longo do livro. Agora, vamos conhecê-lo de modo um pouco

mais formal.

Primeiro, como afirmado na documentação do Node, o File System é um conjunto de wrappers que trabalham com funções POSIX. Isso significa que o módulo consegue acessar sistemas de arquivos obedecendo aos padrões da norma POSIX (que, por si só, é multiplataforma). Esse acesso funciona sem nenhuma adaptação em todos os sistemas operacionais suportados. Portanto, nossas aplicações funcionarão sem modificações no Mac OS X, no Linux e no Windows, bem como em novos ambientes como o Android e em microcontroladores como o Raspberry Pi.

Cada uma das funções do módulo File System vem em duas versões: síncrona e assíncrona, sendo esta última condizente com a natureza assíncrona do Node. Não discutiremos se isso é ou não uma boa ideia, basta aceitarmos que ambas as versões existem e que somo livres para usá-las ou não.

A função assíncrona recebe um callback de tratamento de erros como último argumento, enquanto as funções síncronas imediatamente geram um erro quando uma falha ocorre. Podemos usar a tradicional estrutura de código `try...catch` com as funções síncronas de File System, e nas versões assíncronas da mesma função podemos acessar o erro pelo callback. Nas próximas páginas, nos concentraremos unicamente nas funções assíncronas, mas tenha sempre em mente que existem versões síncronas das mesmas funções.

Além das dezenas de funções, o módulo File System suporta quatro classes:

`fs.FSWatcher`
: Monitora eventos de alteração em arquivos

`fs.ReadStream`
: Um fluxo de leitura

`fs.WriteStream`
: Um fluxo de escrita

`fs.Stats`

Informações devolvidas pelas funções `*stat`

Classe fs.Stats

Quando usamos as funções `fs.stat()`, `fs.lstat()` e `fs.fstat()`, todas elas devolvem um objeto `fs.Stats`, que pode ser usado para verificar se um arquivo (ou uma pasta) existe, além de devolver informações do tipo de entidade (se é um arquivo ou uma pasta), um socket do Unix, as permissões associadas àquele arquivo, quando um objeto foi acessado pela última vez, e assim por diante. O Node oferece algumas funções para acessar informações, como `fs.isFile()` e `fs.isDirectory()`, para determinar se o objeto é um arquivo ou uma pasta. Podemos também acessar diretamente o arquivo que guarda essas estatísticas:

```
var fs = require('fs');
var util = require('util');

fs.stat('./phoenix5a.png', function(err,stats) {
   if (err) return console.log(err);
   console.log(util.inspect(stats));
});
```

Obtemos assim uma estrutura de dados semelhante a esta, em Linux:

```
{ dev: 2048,
  mode: 33204,
  nlink: 1,
  uid: 1000,
  gid: 1000,
  rdev: 0,
  blksize: 4096,
  ino: 1419012,
  size: 219840,
  blocks: 432,
  atime: Thu Oct 22 2015 19:46:41 GMT+0000 (UTC),
  mtime: Thu Oct 22 2015 19:46:41 GMT+0000 (UTC),
  ctime: Mon Oct 26 2015 13:38:03 GMT+0000 (UTC),
  birthtime: Mon Oct 26 2015 13:38:03 GMT+0000 (UTC) }
```

Esse resultado é, basicamente, uma transcrição literal da função `stat()` do POSIX, que devolve as informações de estado e estatísticas de um arquivo. As datas são bastante intuitivas, mas os outros valores podem ser

confusos. O tamanho do arquivo (`size`) está em bytes, `blksize` é o tamanho do bloco no sistema operacional e `blocks` é o número de blocos do objeto. Os dois últimos são `undefined` no Windows.

Um dos componentes mais interessantes é o `mode`. Esse valor contém as permissões do objeto. O problema é que não está completamente legível.

Nesses casos, sempre uso uma função auxiliar ou, no caso do Node, um módulo auxiliar. O stat-mode é um módulo que recolhe os dados estatísticos de objetos devolvidos pela função `fs.stat()` e permite que façamos pesquisas neles. O Exemplo 6.1 demonstra como usá-lo para extrair informações úteis sobre permissões de arquivo.

Exemplo 6.1 – Obtendo permissões de arquivo com o módulo stat-mode

```
var fs = require('fs');
var Mode = require('stat-mode');

fs.stat('./phoenix5a.png', function(err,stats) {
   if (err) return console.log(err);

   // obtém permissões
   var mode = new Mode(stats);

   console.log(mode.toString());
   console.log('Group execute ' + mode.group.execute);
   console.log('Others write ' + mode.others.write);
   console.log('Owner read ' + mode.owner.read);
});
```

Quando executado, o código devolve as seguintes informações para o arquivo fornecido:

```
-rw-rw-r--
Group execute false
Others write false
Owner read true
```

Monitor de sistema de arquivos

Não é nada incomum que uma aplicação fique "escutando" um arquivo ou uma pasta. Quando algum deles é alterado, uma ação é disparada. A classe monitora de sistema de arquivo `fs.FSWatcher` é a interface que lida com isso no Node. Infelizmente, como os desenvolvedores de Node mais

experientes já descobriram, ele tem inconsistências entre as plataformas e não é tão útil assim.

Portanto, vamos ignorá-lo, bem como a função `fs.watch()` que retorna o objeto. Em vez disso, importaremos um módulo de terceiros bem mais útil. Com seus mais de dois milhões de downloads por mês, o Chokidar é um dos módulos de terceiros mais utilizados no mundo do Node (sem contar que faz parte da também popularíssima aplicação Gulp).

Instale o Chokidar usando o comando a seguir (use a chave `-g` para instalar globalmente):

```
npm install chokidar
```

O código a seguir adiciona um monitor (ou watcher, como é conhecido) à pasta atual. Ele verifica mudanças, incluindo alterações dentro dos arquivos. O monitoramento é recursivo e sempre inclui novas pastas contidas dentro da pasta sendo monitorada e qualquer novo arquivo que apareça. O evento bruto (raw) reúne todos os outros eventos, e estes monitoram eventos de nível mais alto.

```
var chokidar = require('chokidar');

var watcher = chokidar.watch('.', {
   ignored: /[\/\\]\./,
   persistent: true
});

var log = console.log.bind(console);

watcher
  .on('add', function(path) { log('File', path, 'has been added'); })
  .on('unlink', function(path) { log('File', path, 'has been removed'); })
  .on('addDir', function(path) { log('Directory', path, 'has been added'); })
  .on('unlinkDir', function(path) {
       log('Directory', path, 'has been removed'); })
  .on('error', function(error) { log('Error happened', error); })
  .on('ready', function() { log('Initial scan complete. Ready for changes.');
})
  .on('raw', function(event, path, details) {
       log('Raw event info:', event, path, details); });

watcher.on('change', function(path, stats) {
  if (stats) log('File', path, 'changed size to', stats.size);
});
```

Se criarmos ou apagarmos arquivos e pastas dentro da pasta sendo monitorada, seremos avisados disso no console. Se algum arquivo mudar de tamanho, também. As funções `unlink()` e `unlinkDir()` refletem o fato de que o "apagamento" aparente do arquivo significa na verdade que ele não está mais vinculado à pasta atual, não aparecendo seu nome de arquivo entre os demais, mas existe no disco e tem vínculos em outros lugares (ou seja, aparecem em outras pastas). Se, por outro lado, esse for o único (ou último) vínculo físico e for apagado, o evento capturado por `unlink()` (ou `unlinkDir()`) indica que o arquivo físico também foi apagado do disco.

Capturar os eventos brutos pode nos devolver informações demais. Mesmo assim, é bastante revelador brincar com eles enquanto aprendemos mais sobre o Chokidar.

Leitura e escrita de arquivos

Inclua o módulo antes de usá-lo:

```
var fs = require('fs');
```

Muitos dos exemplos que usam o módulo File System ao longo do livro empregam os métodos de leitura e escrita que não dependem de fluxos (streams). Há duas maneiras de ler e escrever em um arquivo usando essa funcionalidade que não se baseia em streams.

A primeira é usar os métodos `fs.readFile()` e `fs.writeFile()`, que são, em si, muito simples (ou usar as versões síncronas de ambos). Essas funções abrem o arquivo, executam as operações de leitura e escrita e fecham o arquivo. O código a seguir mostra um arquivo sendo aberto para escrita. Se existir algum conteúdo, ele é truncado. Quando a operação de escrita terminar, o arquivo é novamente aberto, desta vez para leitura, e seu conteúdo é mostrado no console.

```
var fs = require('fs');

fs.writeFile('./some.txt','Writing to a file',function(err) {
   if (err) return console.error(err);
   fs.readFile('./some.txt','utf-8', function(data,err) {
      if (err) return console.error(err);
      console.log(data);
   });
```

```
});
```

Como leitura e escrita no arquivo passam pelo buffer, por padrão, a leitura é feita em UTF-8, por conta da opção `'utf-8'` no segundo argumento de `fs.readFile()`. O buffer poderia também ser convertido em uma string.

O segundo modo de leitura/escrita é abrir um arquivo e criar para ele um descritor (file descriptor, `fd`). Use o descritor de arquivo para escrever e/ou ler nele. A vantagem dessa abordagem é que temos mais controle sobre como o arquivo é aberto e o que podemos fazer com ele depois de aberto.

No código a seguir, um arquivo é criado, algo é escrito nele e depois lido. O segundo parâmetro de `fs.open()` é uma flag que determina quais ações podem ser executadas no arquivo, neste caso um `'a+'`, indicando que o arquivo pode ser aberto para incluir dados nele ou para leitura, criando o arquivo caso não exista. O terceiro parâmetro define as permissões de arquivo (tanto leitura quanto escrita são permitidos).

```
"use strict";

var fs = require('fs');

fs.open('./new.txt','a+',0x666, function(err, fd) {
   if (err) return console.error(err);
   fs.write(fd, 'First line', 'utf-8', function(err,written, str) {
      if (err) return console.error(err);
      var buf = new Buffer(written);
      fs.read(fd, buf, 0, written, 0, function (err, bytes, buffer) {
         if (err) return console.error(err);
         console.log(buf.toString('utf8'));
      });
   });
});
```

O descritor do arquivo é devolvido no callback e então usado com a função `fs.write()`. Uma string é escrita no arquivo, começando na posição 0. Entretanto, observe que, de acordo com a documentação do Node, os dados são sempre inseridos no final do arquivo se estivermos no Linux (o indicador posicional é ignorado), pois o modo de abertura do arquivo foi append (adicionar ao que já existe), indicado pela chave `'a+'`.

A função de callback de `fs.write()` devolve um erro (se ocorrer), o número

de bytes inseridos e a string inserida no final do arquivo. Por fim, `fs.read()` é usado para ler essa linha e a colocar no buffer, que é então mostrado no console.

Obviamente, um aplicativo realmente útil não se preocuparia em ler uma linha de texto que acabamos de gravar, mas o exemplo demonstra os três tipos primários de métodos usados para ler e escrever em um arquivo. Podemos também manipular diretamente uma pasta com o mesmo artifício.

Acesso e administração de pastas

Como já vimos, é possível criar uma pasta, apagá-la e ler os arquivos dentro dela. Também podemos criar um vínculo simbólico[1] para um arquivo e destruir vínculos físicos[2], o que resulta no apagamento do arquivo (desde que não esteja aberto por um programa). Para truncar o arquivo (ou seja, manter sua existência, mas esvaziar seu conteúdo, deixando-o com zero bytes), podemos usar a função `truncate()`.

Para demonstrar algumas das proezas possíveis de fazer em pastas, o código a seguir lista os arquivos na pasta atual e, se algum deles estiver comprimido (com a extensão *.gz*), têm seu link físico destruído por `unlink()`. A tarefa é bastante simplificada pelo uso do módulo Path, que veremos em detalhes em "Acesso a recursos com o módulo Path".

```
'use strict';

var fs = require('fs');
var path = require('path');

fs.readdir ('./',function(err, files) {
   for (let file of files) {
      console.log(file);
      if (path.extname(file) == '.gz') {
         fs.unlink('./' + file);
      }
   }
});
```

Usando fluxos em arquivos

Já colocamos a mão na massa várias vezes com os fluxos de leitura e escrita, mas vale a pena reservar algum tempo para estudá-los um pouco mais a fundo.

Podemos criar fluxos de leitura com `fs.createReadStream()`, informando um caminho e um objeto `options`, ou especificando um descritor de arquivos nas opções e deixando o caminho com um valor nulo. O mesmo se aplica a fluxos de escrita, criados com `fs.createWriteStream()`. Ambos suportam objetos `options`. Por default, o fluxo de leitura foi criado usando as seguintes opções:

```
{ flags: 'r',
  encoding: null,
  fd: null,
  mode: 0o666,
  autoClose: true
}
```

Para usar um descritor de arquivo, basta indicá-lo nas opções. A opção `autoClose` automaticamente fecha o arquivo quando a leitura termina. Se quisermos ler apenas um trecho do arquivo, basta indicar o início e o final (em bytes, com relação ao início do arquivo, byte 0) usando as opções `start` e `end`. Podemos especificar vários tipos de codificação, como, por exemplo `'utf8'`, mas essa codificação pode ser alterada depois com `setEncoding()`.

Fluxos Streams estão documentados em mais de um lugar

A função `setEncoding()`, que podemos usar em um fluxo de leitura criado com `fs.createReadStream()`, está documentada no capítulo sobre o módulo Stream da documentação do Node. Sempre que estiver trabalhando com fluxos, tenha em mente que a documentação a respeito está espalhada em vários capítulos e módulos, e talvez seja preciso um verdadeiro trabalho de garimpo para encontrar o que precisa.

Mais adiante, veremos um código com um exemplo de fluxo de leitura para o módulo File System. Entretanto, antes precisamos conhecer as opções para o fluxo de escrita de File System, criado com `fs.createWriteStream()`. As opções default são:

```
{ flags: 'w',
  defaultEncoding: 'utf8',
  fd: null,
```

```
mode: 0o666 }
```

Novamente, somos livres para usar um descritor de arquivos em vez de um caminho real. Atente também para o fato de que a codificação de texto em um fluxo de escrita é feita com `defaultEncoding`, em vez de `encoding`. Se quiser escrever a partir de uma posição específica depois do começo do arquivo, podemos fazê-lo indicando o valor numérico da posição (em bytes) na opção `start`. A opção `end` não precisa ser especificada porque quem define o tamanho do dado a ser escrito é o próprio dado.

Que tal juntarmos tudo? No Exemplo 6.2, um arquivo é aberto para modificação usando um fluxo de escrita. Na verdade, a única modificação é a inserção de uma string em um ponto específico do arquivo. Usaremos um descritor de arquivos para este exemplo, portanto quando a aplicação chama `fs.createWriteStream()`, não inicia a abertura de arquivos no mesmo momento em que cria o fluxo de escrita. Não execute este código ainda, apenas o salve.

Exemplo 6.2 – Modificando um arquivo existente pela inclusão de uma string

```
var fs = require('fs');

fs.open('./working.txt', 'r+',function (err, fd) {
   if (err) return console.error(err);

   var writable = fs.createWriteStream(null,{fd: fd, start: 10,
                                       defaultEncoding: 'utf8'});

writable.write(' inserting this text ');
});
```

Observe que o arquivo foi aberto com a flag `r+`, permitindo que a aplicação possa ler e modificar esse arquivo.

No Exemplo 6.3, o mesmo arquivo é aberto, mas desta vez o conteúdo será lido. A flag usada é `r`, pois o arquivo deve ser apenas lido. O código lê todo o conteúdo do arquivo neste exemplo.

A codificação de texto é alterada para `utf8` usando `setEncoding()`. No Exemplo 6.2, visto anteriormente, mudamos a codificação para `utf8` quando ajustamos a flag `defaultEncoding`.

Exemplo 6.3 – Lendo o conteúdo de um arquivo usando um fluxo

```
var fs = require('fs');

var readable =
   fs.createReadStream('./working.txt').setEncoding('utf8');

var data = '';
readable.on('data', function(chunk) {
   data += chunk;
});

readable.on('end', function() {
   console.log(data);
});
```

Rodar essa aplicação de leitura antes e depois da anterior, que modifica o arquivo, mostra a alteração feita. Na primeira execução, a aplicação de leitura devolve o conteúdo inalterado de *working.txt*:

```
Now let's pull this all together, and read and write with a stream.
```

Na segunda execução, temos:

```
Now let's inserting this text and read and write with a stream.
```

Pensando um pouco, chegamos à conclusão de que economizaríamos bastante tempo se pudéssemos abrir o arquivo para leitura e depois redirecionássemos os resultados para um fluxo de escrita. Poderíamos facilmente usar a função `pipe()`, disponível para todos os fluxos de leitura. Entretanto, não há como modificar os resultados no meio do caminho porque o fluxo de escrita é apenas isso: de escrita. Não é um fluxo duplex, muito menos de transformação. Contudo, se o objetivo for copiar o conteúdo de cá para lá, funciona muito bem.

```
var fs = require('fs');

var readable =
   fs.createReadStream('./working.txt');

var writable = fs.createWriteStream('./working2.txt');

readable.pipe(writable);
```

Veremos um fluxo de transformação mais adiante, em “Compactação é com o ZLib”.

Acesso a recursos com o módulo Path

O módulo Path, considerado um utilitário do Node, existe para facilitar a transformação e extração de dados pelos caminhos do sistema de arquivos. Ele também oferece uma maneira neutra (agnóstica quanto ao ambiente) de lidar com os caminhos nos sistemas de arquivos, assim não é necessário escrever um módulo para o Linux e outro para o Windows.

Já vimos o recurso de extração quando obtivemos a extensão de um arquivo enquanto pesquisávamos os arquivos em uma pasta:

```
'use strict';

var fs = require('fs');
var path = require('path');
fs.readdir ('./',function(err, files) {
   for (let file of files) {
      console.log(file);
      if (path.extname(file) == '.gz') {
         fs.unlink('./' + file);
      }
   }
});
```

Se o que queremos é obter o nome do arquivo, sem a extensão, o código apropriado é:

```
'use strict';

var fs = require('fs');
var path = require('path');

fs.readdir ('./',function(err, files) {
   for (let file of files) {
      let ext = path.extname(file);
      let base = path.basename(file, ext);
      console.log ('file ' + base + ' with extension of ' + ext);
   }
});
```

O segundo argumento da função `path.basename()` faz com que apenas o nome principal do arquivo seja devolvido, sem a extensão.

Um exemplo da neutralidade ambiental do módulo Path é a propriedade `path.delimeter`. Esse é o delimitador, dentro da variável de ambiente `PATH`,

que separa os caminhos uns dos outros. O delimitador é específico para cada sistema operacional: no Linux é o sinal de dois-pontos (:), enquanto no Windows é o ponto e vírgula (;). Se quisermos uma lista que desmembre cada valor da variável de ambiente `PATH` para que uma aplicação possa usar, temos de empregar `path.delimeter`:

```
var path = require('path');

console.log(process.env.PATH);
console.log(process.env.PATH.split(path.delimiter));
```

Agora a aplicação consegue funcionar em ambos os ambientes. A última linha retorna um array em que cada elemento é um dos caminhos que estavam na variável de ambiente `PATH`.

Outra diferença é o caractere de nível de pasta no sistema de arquivo. Alguns sistemas usam a barra (/), outros a barra invertida (\). No Capítulo 5, criamos um servidor de arquivos simples que chamava de um caminho no sistema de arquivos o recurso a ser servido. No Windows, os caminhos usam a barra invertida, mas no Linux empregam a barra normal. Fizemos a aplicação funcionar em ambos os ambientes passando a string de caminho pelo método `path.normalize()`:

```
pathname = path.normalize(base + req.url);
```

O grande trunfo do módulo Path não é exatamente as maravilhosas transformações de string que ele faz: afinal, poderíamos fazer o mesmo com um objeto `String` ou mesmo com expressões regulares. O grande diferencial desse módulo é que os caminhos do sistema de arquivos são transformados de forma agnóstica (ou seja, tanto faz o sistema operacional, dará certo sempre).

Se quisermos analisar um caminho e dividi-lo em seus componentes, a função `path.parse()` vem ao nosso auxílio. Os resultados diferem significativamente, dependendo do sistema operacional. No Windows, quando usamos `require.main.filename` (ou, de forma abreviada, `__filename`), que é uma propriedade que contém o caminho e o nome da aplicação sendo executada, temos:

```
{ root: 'C:\\',
  dir: 'C:\\Users\\Shelley',
```

```
  base: 'work',
  ext: '.js',
  name: 'path1' }
```

No Ubuntu, temos:

```
{ root: '/',
  dir: '/home/examples/public_html/learnnode2',
  base: 'path1.js',
  ext: '.js',
  name: 'path1' }
```

Criando um comando para usar no terminal

Em ambientes Unix, podemos facilmente criar aplicações em Node que rodam direto no shell, sem precisar chamar o comando `node` seguido do nome da aplicação.

No Windows também dá

Para criar um comando no Windows que rode sua aplicação diretamente, sem precisar digitar Node antes do nome do programa, basta criar um arquivo de lote (extensão .BAT). Dentro do arquivo de lote, coloqueo mesmo comando Node que você usaria normalmente para chamá-la: node nome_aplicação.

Para demonstrar, usarei o módulo Commander, que já vimos no Capítulo 3, e um *processo-filho* para acessar um programa externo chamado ImageMagick, uma ferramenta de tratamento de imagens muito poderosa.

Processo-filho

Os processos-filhos serão discutidos no Capítulo 8.

Em minha aplicação, uso o ImageMagick para pegar uma imagem existente e adicionar a ela um efeito Polaroid, gravando o resultado como um novo arquivo. Como mostrado no Exemplo 6.4, o Commander faz todo o processamento das opções informadas ao nosso programa e ainda providencia um texto de ajuda.

Exemplo 6.4 – O Node como uma aplicação de linha de comando

```
#!/usr/bin/env node

var spawn = require('child_process').spawn;
```

```
var program = require('commander');

program
   .version ('0.0.1')
   .option ('-s, --source [file name]', 'Source graphic file name')
   .option ('-f, --file [file name]', 'Resulting file name')
   .parse(process.argv);

if ((program.source === undefined) || (program.file === undefined)) {
   console.error('source and file must be provided');
   process.exit();
}

var photo = program.source;
var file = program.file;

// array de conversão
var opts = [
photo,
"-bordercolor", "snow",
"-border", "20",
"-background","gray60",
"-background", "none",
"-rotate", "6",
"-background", "black",
"(", "+clone", "-shadow", "60x8+8+8", ")",
"+swap",
"-background", "none",
"-thumbnail", "240x240",
"-flatten",
file];

var im = spawn('convert', opts);
im.stderr.on('data', (data) => {
   console.log(`stderr: ${data}`);
});

im.on('close', (code) => {
   console.log(`child process exited with code ${code}`);
});
```

Para converter um código-fonte comum em um comando que funciona no terminal, basta colocar a seguinte linha no topo do arquivo:

```
#!/usr/bin/env node
```

Os caracteres `#!` são apelidados, em inglês, como *shebang* (numa tradução aproximada e bastante livre, “cabum”). Depois do shebang, é necessário

colocar qual interpretador de linguagem será usado para executar o código que está logo abaixo, dentro do arquivo. Em nosso caso, é o Node. O subdiretório é o caminho onde a aplicação reside.

O arquivo deve ser salvo com a extensão *.js*. No Unix/Linux/Mac OS X, o arquivo precisa também ser convertido para executável, senão comporta-se apenas como um arquivo de texto. Isso é feito com o comando `chmod`:

```
chmod a+x polaroid
```

Para rodar nosso programa no terminal:

```
./polaroid -h
```

A chave `-h` mostra a ajuda do programa (gerada pelo Commander). Já o comando:

```
./polaroid -s phoenix5a.png -f phoenix5apolaroid.png
```

cria uma nova imagem, a partir da antiga, com o efeito Polaroid aplicado. Infelizmente, o utilitário não funciona no Windows[3], mas nos sistemas suportados os resultados são muito bons.

Criar um utilitário ou programa de linha de comando não é a mesma coisa que criar uma aplicação independente. Esta última implica que poderíamos instalar a aplicação sem precisar que o Node (ou qualquer outra dependência) esteja previamente instalado.

Criando uma aplicação independente com o NW.js

Na época em que este livro foi escrito, a única funcionalidade conhecida para criar aplicações independentes em Node era o NW.js, da Intel (antes conhecido como node-webkit). Podemos usá-lo para empacotar todos os arquivos e distribuir o pacote como uma aplicação completa. O NW.js providencia todos os requisitos e dependências para que o conjunto funcione.

Compactação é com o ZLib

O módulo ZLib oferece funcionalidade de compactação. Ele também é baseado em um fluxo de transformação, que fica aparente quando consultamos o exemplo oferecido pela documentação do Node para compactar um arquivo. O exemplo a seguir é uma leve modificação daquele, para poder trabalhar com arquivos maiores.

```
var zlib = require('zlib');
```

```
var fs = require('fs');

var gzip = zlib.createGzip();

var inp = fs.createReadStream('test.png');
var out = fs.createWriteStream('test.png.gz');
inp.pipe(gzip).pipe(out);
```

O fluxo de entrada está diretamente conectado à saída, com o compactador gzip no meio, transformando o conteúdo – neste caso, uma imagem em formato PNG.

O ZLib suporta os algoritmos de compactação `zlib` e `deflate`, este último mais complexo e controlável. Observe que, ao contrário de `zlib`, no qual podemos descompactar o arquivo criado por nossa aplicação usando um comando externo como o utilitário gunzip (ou unzip) do Linux, não temos essa oportunidade com `deflate`. Precisamos usar o Node ou algo semelhante que trabalhe com `deflate` para descompactar o arquivo.

Para demonstrar como compactar e descompactar um arquivo, criaremos dois comandos de terminal, chamados de compress e uncompress. O primeiro compactará o arquivo usando `gzip` ou `deflate`. Como precisamos de opções no comando para escolher o algoritmo de compactação, empregaremos o módulo Commander:

```
var zlib = require('zlib');
var program = require('commander');
var fs = require('fs');

program
   .version ('0.0.1')
   .option ('-s, --source [file name]', 'Source File Name')
   .option ('-f, --file [file name]', 'Destination File name')
   .option ('-t, --type <mode>', /^(gzip|deflate)$/i)
   .parse(process.argv);

var compress;
if (program.type == 'deflate')
   compress = zlib.createDeflate();
else
   compress = zlib.createGzip();

var inp = fs.createReadStream(program.source);
var out = fs.createWriteStream(program.file);

inp.pipe(compress).pipe(out);
```

Esse utilitário é muito interessante e (como o nome já diz) útil, ainda mais no Windows, que não tem acesso a esse tipo de compactação. Entretanto, um uso bastante popular para as tecnologias de compactação é nas solicitações web. A documentação do Node contém inúmeros exemplos de uso do ZLib nessa situação. Há também vários exemplos de recuperação via web de um arquivo compactado, usando o módulo (que vimos no Capítulo 5) e a função `http.request()`.

Em vez de solicitar um arquivo comprimido, vamos enviar um arquivo comprimido para um servidor que o descompacta ao receber. Os exemplos de servidor e cliente são adaptados dos Exemplos 5.1 e 5.2, mas o código foi modificado para compactar um arquivo PNG enorme e o enviar em uma solicitação HTTP. O servidor, ao receber o arquivo, o descompacta e grava no disco.

O código do servidor está no Exemplo 6.5. Observe que os dados sendo enviados são recebidos como um array de lascas (chunks), que eventualmente são usadas para criar um Buffer usando `buffer.concat()`. Como estamos lidando com um buffer e não com um stream, não podemos usar a função `pipe()`. Em vez disso, empregamos a função de conveniência do ZLib, `zlib.unzip`, passando a ela o Buffer e uma função de callback, que tem por argumentos um erro e um resultado. O resultado também é um Buffer, cujo conteúdo é inserido em um fluxo de escrita com a função `write()`. Para garantir que o arquivo original não seja sobrescrito, acrescentamos a data e a hora da conversão ao nome do novo arquivo.

Exemplo 6.5 – Criando um servidor web que aceita dados compactados e os descompacta em um arquivo

```
var http = require('http');
var zlib = require('zlib');
var fs = require('fs');

var server = http.createServer().listen(8124);

server.on('request', function(request,response) {

   if (request.method == 'POST') {
      var chunks = [];
```

```
      request.on('data', function(chunk) {
         chunks.push(chunk);
      });

      request.on('end', function() {
         var buf = Buffer.concat(chunks);
         zlib.unzip(buf, function(err, result) {
            if (err) {
               response.writeHead(500);
               response.end();
               return console.log('error ' + err);
            }
            var timestamp = Date.now();
            var filename = './done' + timestamp + '.png';
            fs.createWriteStream(filename).write(result);
         });
         response.writeHead(200, {'Content-Type': 'text/plain'});
         response.end('Received and undecompressed file\n');
      });
   }
});

console.log('server listening on 8214');
```

A chave do sucesso no código do cliente, mostrado no Exemplo 6.6, é garantir a codificação correta do conteúdo (campo `Content-Encoding` no cabeçalho HTTP), que deve ser `'gzip,deflate'`. O `Content-Type` (tipo de conteúdo) também foi alterado para `'application/javascript'`.

Exemplo 6.6 – Cliente que compacta um arquivo e o redireciona para uma solicitação web

```
var http = require('http');
var fs = require('fs');
var zlib = require('zlib');

var gzip = zlib.createGzip();

var options = {
   hostname: 'localhost',
   port: 8124,
   method: 'POST',
   headers: {
      'Content-Type': 'application/javascript',
      'Content-Encoding': 'gzip,deflate'
   }
```

```
};
var req = http.request(options, function(res) {
   res.setEncoding('utf8');
   var data = '';
   res.on('data', function (chunk) {
      data+=chunk;
   });
   res.on('end', function() {
      console.log(data)
   })
});
req.on('error', function(e) {
   console.log('problem with request: ' + e.message);
});
// envia para o servidor o arquivo compactado em gzip
var readable = fs.createReadStream('./test.png');
readable.pipe(gzip).pipe(req);
```

O cliente abre o arquivo a ser compactado e o redireciona (com `pipe()`) a um fluxo de transformação do ZLib, que por sua vez redireciona o resultado à solicitação web (que é, em si, um fluxo de escrita). Estamos usando unicamente fluxos nesse código, assim podemos utilizar a função `pipe()` à vontade. Não podemos usá-la no servidor porque os dados são transmitidos como lascas em formato Buffer.

Nosso código guarda na memória RAM o arquivo recebido. Se a aplicação começar a escalonar para cima (ou seja, o número de acessos e o tamanho dos arquivos está em crescimento incontrolável, devido à popularidade da aplicação), teremos um problema sério. Para contorná-lo, o ideal seria gravar temporariamente o arquivo descompactado, compactá-lo e depois apagar o arquivo temporário. Deixo esse exercício como um desafio para o leitor.

Readline e os redirecionamentos com pipe()

Temos usado os redirecionamentos com `pipe()` desde o Capítulo 5. Uma das demonstrações mais simples do `pipe()` é abrir uma sessão do REPL e digitar o seguinte:

```
> process.stdin.resume();
> process.stdin.pipe(process.stdout);
```

A partir daqui tudo o que for digitado no REPL será ecoado na tela.

Para manter o fluxo de saída aberto, e assim receber dados continuamente, basta passar a opção `{ end: false }` a esse fluxo:

```
process.stdin.pipe(process.stdout, { end : false });
```

O processamento linha a linha do REPL é na realidade implementado usando o último módulo nativo que veremos neste capítulo: Readline. O simples fato de importar o Readline inicia uma thread de comunicação sem fim. Para incluir o módulo Readline, faça:

```
var readline = require('readline');
```

Tenha em mente que, uma vez incluído esse módulo, o programa do Node não é encerrado até que fechemos a interface.

A documentação no site do Node contém um exemplo de como iniciar e encerrar uma interface Readline, que aparece levemente adaptado no Exemplo 6.7. A aplicação faz uma pergunta imediatamente após iniciada e em seguida mostra a resposta. Ela também monitora qualquer "comando", que na verdade é qualquer coisa que você digite e que termine com `\n`. Se o comando for `.leave`, a aplicação é encerrada, caso contrário apenas repete os comandos e pede ao usuário que digite mais. Um Ctrl-C ou Ctrl-D também causam o encerramento da aplicação, embora de forma não tão elegante.

Exemplo 6.7 – Usando Readline para criar uma interface de usuário simples, operada por comandos

```
var readline = require('readline');

// cria a interface
var rl = readline.createInterface(process.stdin, process.stdout);

// pergunta
rl.question(">>What is the meaning of life? ", function(answer) {
   console.log("About the meaning of life, you said " + answer);
   rl.setPrompt(">> ");
   rl.prompt();
});

// função para fechar a interface
```

```
function closeInterface() {
   rl.close();
   console.log('Leaving Readline');
}

// monitora a digitação do comando .leave
rl.on('line', function(cmd) {
   if (cmd.trim() == '.leave') {
      closeInterface();
      return;
   }
   console.log("repeating command: " + cmd);
   rl.prompt();
});

rl.on('close', function() {
   closeInterface();
});
```

Uma sessão em nosso interpretador se pareceria com isto:

```
>>What is the meaning of life? ===
About the meaning of life, you said ===
>>This could be a command
repeating command: This could be a command
>>We could add eval in here and actually run this thing
repeating command: We could add eval in here and actually run this thing
>>And now you know where REPL comes from
repeating command: And now you know where REPL comes from
>>And that using rlwrap replaces this Readline functionality
repeating command: And that using rlwrap replaces this Readline functionality
>>Time to go
repeating command: Time to go
>>.leave
Leaving Readline...
Leaving Readline...
```

Parece familiar, não? No Capítulo 4, usamos `rlwrap` para nos sobrepor à funcionalidade de linha de comando do REPL. Usamos esta linha para dispará-lo:

```
env NODE_NO_READLINE=1 rlwrap node
```

Agora, sabemos o que aquela opção estava disparando: com ela, o REPL ignora o módulo Readline que vem no Node e usa `rlwrap` em seu lugar.

---

1 N. do T.: Desenvolvedores que vêm do Windows e não têm muito acesso ao sistema de arquivos do servidor (na maioria das vezes, Linux) podem não estar familiarizados com a ideia de vínculo simbólico, também conhecido como link simbólico ou symlink. É um elemento bastante comum nos Unix (o que inclui Linux e Mac OS X). Poderíamos pensar nele como um atalho, parecido com o do Windows em modo gráfico, que aponta para uma pasta ou um arquivo real que estejam em outro lugar do disco. A diferença é que os atalhos do Windows não passam de arquivos de texto com extensão .LNK que guardam as informações do arquivo "atalhado". Já os symlinks do Unix são estruturas intrínsecas do sistema de arquivos e não precisam de um arquivo auxiliar; fora isso seu comportamento é idêntico ao dos atalhos do Windows. A partir do Windows Vista, também é possível criar symlinks no sistema da Microsoft, mas apenas na janela de terminal – via interface gráfica, apenas os velhos atalhos .LNK são permitidos.

2 N. do T.: Além do vínculo simbólico há o vínculo físico, ou hard link, que, em vez de atalho, comporta-se como se o mesmo arquivo ou pasta estivesse em dois (ou mais) lugares ao mesmo tempo. Apagar um symlink não afeta o arquivo (ou a pasta) em disco, pois é só um atalho. Contudo, se o arquivo tiver mais de um hard link, ele continuará existindo mesmo que apaguemos alguns deles, até que o último hard link seja apagado. Nesse momento, o arquivo é destruído. No Windows 2000 em diante é possível criar e apagar hard links (chamados de junction points no jargão da Microsoft), mas apenas na janela de terminal e somente para pastas, nunca para arquivos.

3 N. do T.: Existe uma versão oficial do ImageMagick para Windows, o que é indício de que talvez o código possa ser adaptado para rodar também nessa plataforma. Mais informações em *pt.wikipedia.org/wiki/ImageMagick*, *tinyurl.com/SaraujoImageMagik* e *www.imagemagick.org*.

CAPÍTULO 7

Redes, sockets e segurança

O núcleo de uma aplicação em Node se apoia invariavelmente em dois componentes de infraestrutura principais: redes e segurança. Não podemos falar de rede sem também discutir sockets.

Costumo reunir redes e segurança no mesmo grupo porque, quando deixamos de ter uma única máquina isolada e passamos a ter duas ou mais, a segurança deve ocupar seus pensamentos a todo momento. Cada vez que terminamos um componente de alguma aplicação, a primeira pergunta que precisamos nos fazer é: isto aqui está seguro? De nada adianta um código tão belo que beira à poesia se ele deixa entrar todo tipo de sujidade na máquina do usuário.

Servidores, fluxos e sockets

Grande parte da API central do Node está relacionada à criação de serviços que monitoram (ou, no jargão aceito pelo pessoal de redes, "escutam") tipos específicos de comunicação. Nos Capítulos 1 e 5 usamos o módulo HTTP para criar servidores web que "escutavam" solicitações HTTP. Outros módulos podem criar um servidor para o protocolo Transmission Control Protocol (TCP), ou usar criptografia pelo protocolo Transport Layer Security (TLS), ou ainda um socket para o protocolo User Datagram Protocol (UDP). Discutiremos sobre TLS mais adiante neste capítulo, mas por enquanto vamos entender como é a funcionalidade do Node para TCP e UDP. Antes, descobriremos o que são esses tais sockets.

Sockets e fluxos

Um *socket* é um terminal de comunicação, e um *socket de rede* é um terminal de comunicação entre dois computadores diferentes ligados em rede. Os dados fluem entre os sockets no que já conhecemos pelo nome de *stream*, ou fluxo em português. Os dados num fluxo podem ser transmitidos em formato binário em um buffer, ou em formato Unicode em uma string. Ambos os tipos de dados são transmitidos como *pacotes*: os dados são divididos em pedacinhos pequenos e de igual tamanho. Há um tipo especial de pacote, chamado pacote de término (FIN)[1], que é enviado ao socket para sinalizar o encerramento da transmissão.

Imagine duas pessoas conversando por um rádio intercomunicador. Os rádios são os terminais (em inglês, endpoints), ou seja, são os *sockets* da comunicação. Quando duas ou mais pessoas querem falar entre si, precisam sintonizar no mesmo canal. Uma delas pressiona o botão no rádio e se conecta a uma das pessoas usando alguma forma de identificação. Quando alguém termina de falar e quer informar aos demais que já acabou e que está escutando, diz "câmbio". A outra pessoa então pressiona o botão em seu rádio, confirma que recebeu a comunicação e também diz "câmbio" para indicar que está na escuta. A comunicação continua até que um dos participantes diz "câmbio e desligo", o que significa que a conversa chegou ao fim. Apenas uma pessoa pode falar por vez, e é por isso que esse tipo de comunicação por rádio é conhecida como *half-duplex* (em português, semiduplex), porque a comunicação só pode ocorrer em uma direção por vez. Já a comunicação *full-duplex* permite escutar e falar ao mesmo tempo, ou seja, há transmissão e recepção simultâneas nos dois sentidos.

O conceito também se aplica aos fluxos (streams) do Node. Já trabalhamos com fluxos half-duplex e full-duplex no Capítulo 6. Os fluxos usados para escrever e ler a partir de arquivos eram exemplos de comunicação half-duplex: eles permitiam ler de um arquivo ou gravar em um arquivo, mas jamais as duas coisas ao mesmo tempo. Por outro lado, o fluxo de compactação da biblioteca zlib é um exemplo de fluxo full-duplex, pois permite leitura e gravação simultâneas.

Agora aplicaremos toda essa bagagem que aprendemos com os arquivos

em fluxos de rede (TCP) e de criptografia (Crypto). Aprenderemos sobre Crypto mais adiante neste capítulo. Primeiro, vamos conhecer mais sobre o TCP.

Sockets e servidores TCP

O TCP oferece uma plataforma de comunicação para que o Node possa suportar aplicações de internet, como web services ou email. O módulo permite transmitir dados entre os sockets do cliente e do servidor de forma confiável. O TCP propicia a infraestrutura sobre a qual a camada de aplicação reside. Um exemplo é o HTTP.

Podemos criar um servidor e um cliente TCP da mesma forma como fizemos com o módulo HTTP, com algumas diferenças. Ao criar um servidor TCP, em vez de passar um `requestListener` para a função de criação do servidor, com seus objetos separados de resposta e solicitação, o único argumento da função de callback do TCP é uma instância de um socket que pode tanto receber quanto enviar dados.

Para demonstrar como o TCP funciona, o Exemplo 7.1 contém o código que cria um servidor TCP. Depois que o socket do servidor é criado, passa a monitorar ("escutar") dois eventos: o momento em que um bloco de dados é recebido e o momento em que o cliente fecha a conexão. O servidor então mostra os dados no console e devolve esses mesmos dados para o cliente.

O servidor TCP também anexa um descritor para os eventos `listening` e `error`. Nos capítulos anteriores, simplesmente mostramos uma mensagem com `console.log()` depois que cada servidor foi criado, o que é uma prática bastante comum no Node. Entretanto, como o evento `listen()` é assíncrono, tecnicamente essa prática é incorreta – a mensagem seria impressa antes que o evento de escuta fosse iniciado. Em vez disso, podemos incorporar a mensagem na função de callback chamada pela função `listen` ou, ainda, (e é o que fizemos no exemplo) anexar um descritor ao evento `listening` e gerar a mensagem no momento exato.

O tratamento de erros também é mais sofisticado nesse exemplo e foi criado segundo o recomendado na documentação do Node. A aplicação

processa um evento `error` e, se o erro tiver sido gerado porque a porta indicada já está em uso, espera um determinado período de tempo e tenta novamente. Para outros erros – como acessar uma porta como a 80, que requer privilégios especiais – a mensagem completa de erro é mostrada no console.

Exemplo 7.1 – Um servidor TCP bastante simples, com um socket aguardando conexões do cliente na porta 8124

```
var net = require('net');
const PORT = 8124;

var server = net.createServer(function(conn) {
   console.log('connected');

   conn.on('data', function (data) {
      console.log(data + ' from ' + conn.remoteAddress + ' ' +
         conn.remotePort);
      conn.write('Repeating: ' + data);
   });

   conn.on('close', function() {
      console.log('client closed connection');
   });

}).listen(PORT);

server.on('listening', function() {
   console.log('listening on ' + PORT);
});

server.on('error', function(err){
   if (err.code == 'EADDRINUSE') {
      console.warn('Address in use, retrying...');
      setTimeout(() => {
         server.close();
         server.listen(PORT);
      }, 1000);
   }
   else {
      console.error(err);
   }
});
```

Ao criar o socket TCP, passamos um objeto opcional contendo parâmetros, consistindo de dois valores: `pauseOnConnect` e `allowHalfOpen`. O

valor default para ambos é `false`:

```
{ allowHalfOpen: false,
  pauseOnConnect: false }
```

Alterar `allowHalfOpen` para `true` instrui o socket para não enviar um pacote FIN quando receber um FIN vindo do cliente. Isso faz com que o socket continue aberto para escrita (mas não para leitura). Para fechar o socket, nesse caso, é necessário usar a função `end()`. Já alterar `pauseOnConnect` para `true` faz com que a conexão seja iniciada, mas nenhum dado será lido. Para iniciar a leitura dos dados, é necessário chamar o método `resume()` no socket.

Para testar o servidor, podemos usar um cliente TCP existente, como o utilitário netcat (nc) no Linux ou no OS X, ou um programa equivalente no Windows. Com o netcat, o comando a seguir se conecta como cliente à nossa aplicação (que está agindo como servidor) na porta 8124, transferindo para o servidor dados de um arquivo-texto que está no cliente:

```
nc burningbird.net 8124 < mydata.txt
```

No Windows, existem ferramentas para trabalhar com conexões TCP, como o SocketTest, que podemos usar para testes.

Em vez de usar uma ferramenta para testar o servidor, podemos criar nossa própria aplicação cliente. Nosso cliente TCP é tão simples quanto o servidor e é mostrado no Exemplo 7.2. Os dados são transmitidos como um buffer, mas podemos usar `setEncoding()` para lê-los como uma string em `utf8`. O método `write()` do socket é usado para transmitir os dados. A aplicação cliente também adiciona funções de escuta a dois eventos: `data`, para dados recebidos, e `close`, caso o servidor feche a conexão.

Exemplo 7.2 – O socket do cliente, que envia dados ao servidor TCP

```
var net = require('net');
var client = new net.Socket();
client.setEncoding('utf8');

// conecta ao servidor
client.connect ('8124','localhost', function () {
   console.log('connected to server');
```

```
    client.write('Who needs a browser to communicate?');
});

// ao receber dados, reenviá-los ao servidor
process.stdin.on('data', function (data) {
    client.write(data);
});

// ao receber os dados devolvidos, enviá-los ao console
client.on('data',function(data) {
    console.log(data);
});

// quando o servidor fechar a conexão
client.on('close',function() {
    console.log('connection is closed');
});
```

Os dados que serão transmitidos entre os dois sockets devem ser digitados no terminal do cliente e só são transmitidos quando pressionamos Enter. A aplicação cliente envia o texto digitado ao servidor TCP, que o escreve no console assim que chega. O servidor então devolve a mesma mensagem para o cliente, que por sua vez mostra a mensagem em seu próprio console. O servidor mostra ainda, no console, o endereço IP e a porta do cliente, obtidos pelas propriedades `remoteAddress` e `remotePort` do socket. O exemplo a seguir mostra a saída no console do servidor depois do envio de algumas strings pelo cliente:

```
Hey, hey, hey, hey-now.
from ::ffff:127.0.0.1 57251
Don't be mean, we don't have to be mean.
from ::ffff:127.0.0.1 57251
Cuz remember, no matter where you go,
from ::ffff:127.0.0.1 57251
there you are.
from ::ffff:127.0.0.1 57251
```

A conexão entre o cliente e o servidor é mantida aberta até que um dos dois seja encerrado usando Ctrl-C. O socket que ainda estiver aberto receberá um evento `close` que é ecoado para o console. O servidor pode também servir múltiplas conexões vindas de diversos clientes, pois todas as funções relevantes são assíncronas.

Endereços IPv4 mapeados para IPv6

A sessão de exemplo entre o cliente e o servidor TCP mostra um endereço IPv4 mapeado para IPv6, com a adição do prefixo `::ffff`.

Em vez de nos conectar a uma porta com o servidor TCP, podemos nos conectar diretamente a um socket. Para demonstrar esse recurso, modifiquei o servidor TCP dos exemplos anteriores, fazendo com que o servidor se vincule a um socket do Unix em vez de uma porta, como mostrado no Exemplo 7.3. O *socket Unix* é um caminho do sistema de arquivos existente em algum lugar do servidor. As permissões de escrita e leitura podem ser usadas para controlar de forma limitada o acesso à aplicação, o que torna essa técnica mais vantajosa do que um socket de internet.

Também tive de modificar o tratamento de erros para destruir o vínculo (isto é, apagar o arquivo com unlink) para o socket Unix caso este esteja em uso quando a aplicação é iniciada. Em ambiente de produção, é preciso garantir que nenhum outro cliente esteja usando o socket antes de apagá-lo de forma tão brusca.

Exemplo 7.3 – Um servidor TCP vinculado a um socket do Unix

```
var net = require('net');
var fs = require('fs');

const unixsocket = '/somepath/nodesocket';

var server = net.createServer(function(conn) {
   console.log('connected');

   conn.on('data', function (data) {
      conn.write('Repeating: ' + data);
   });

   conn.on('close', function() {
      console.log('client closed connection');
   });

}).listen(unixsocket);

server.on('listening', function() {
   console.log('listening on ' + unixsocket);
});

// se o programa for reiniciado, o socket deve ser apagado (com unlink)
```

```
server.on('error',function(err) {
   if (err.code == 'EADDRINUSE') {
      fs.unlink(unixsocket, function() {
         server.listen(unixsocket);
      });
   } else {
      console.log(err);
   }
});

process.on('uncaughtException', function (err) {
   console.log(err);
});
```

Também usei `process` como proteção extra contra qualquer exceção que seja gerada e não seja administrada pela aplicação.

Verifique se existe outra instância do servidor em execução

Antes de apagar o socket com unlink, verifique se existe outra instância ativa do servidor. No site Stack Overflow, encontrei uma solução que oferece uma técnica alternativa de limpeza para essa situação.

A aplicação cliente é mostrada no Exemplo 7.4. Não é muito diferente do cliente anterior que usamos para conectar pela porta TCP. A única diferença é o ajuste do ponto de conexão.

Exemplo 7.4 – Conectando ao socket do Unix e mostrando os dados recebidos

```
var net = require('net');
var client = new net.Socket();
client.setEncoding('utf8');

// conecta ao servidor
client.connect ('/somepath/nodesocket', function () {
   console.log('connected to server');
   client.write('Who needs a browser to communicate?');
});

// ao receber dados, reenviá-los ao servidor
process.stdin.on('data', function (data) {
   client.write(data);
});

// ao recebê-los de volta, enviar ao console
client.on('data',function(data) {
```

```
   console.log(data);
});

// quando o servidor for encerrado, faça:
client.on('close',function() {
   console.log('connection is closed');
});
```

"Sentinelas nos portões", discorre sobre a versão SSL do protocolo HTTP, conhecida como HTTPS, bem como sobre criptografia no geral e os protocolos TLS/SSL.

Sockets para datagramas UDP

O TCP precisa de uma conexão dedicada entre os dois terminais de comunicação. Já o UDP é um protocolo não orientado a conexão, o que significa que não há garantias de que os dois terminais realmente conversarão entre si. Por essa razão, o UDP é menos confiável e robusto que o TCP. Por outro lado, o UDP é geralmente mais rápido que o TCP, o que o torna muito popular para serviços que precisem de respostas rápidas ou em tempo real, como a telefonia IP (mais conhecida como Voice over Internet Protocol, ou VoIP), em que os requisitos de confiabilidade de uma conexão TCP degradariam a qualidade do áudio.

O núcleo do Node suporta ambos os tipos de sockets. Já vimos como isso funciona no TCP, agora é a vez do UDP.

O módulo para trabalhar com UDP se chama `dgram`:

```
require ('dgram');
```

Para criar um socket UDP, use o método `createSocket`, informando o tipo – pode ser `udp4` ou `udp6`. Podemos também passar um callback que monitora eventos. Ao contrário das mensagens enviadas por TCP, quando empregamos o protocolo UDP as mensagens precisam ser enviadas em buffers, jamais em strings.

O Exemplo 7.5 contém o código para um cliente UDP simples. Nele, os dados são acessados por `process.stdin` e enviados como estão pelo socket UDP. Observe que não definimos a codificação da string, pois o UDP só aceita um buffer, e os dados em `process.stdin` *estão* em um buffer. Entretanto, é preciso converter o buffer em string com o método `toString`

do próprio buffer, para que possamos ecoar a mensagem pelo `console.log()`.

Exemplo 7.5 – Um cliente de datagrama que envia mensagens digitadas no terminal

```
var dgram = require('dgram');

var client = dgram.createSocket("udp4");

process.stdin.on('data', function (data) {
   console.log(data.toString('utf8'));
   client.send(data, 0, data.length, 8124, "examples.burningbird.net",
      function (err, bytes) {
         if (err)
            console.error('error: ' + err);
         else
            console.log('successful');
   });
});
```

O servidor UDP, mostrado no Exemplo 7.6, é ainda mais simples que o cliente. Tudo o que a aplicação servidora faz é criar um socket, associá-lo a uma porta específica (8124) e monitorar eventos `message`. Quando uma mensagem chega, a aplicação a envia para o console, bem como o endereço IP e a porta do emissor, usando nosso velho conhecido `console.log`. Não é necessário codificar o texto para exibir a mensagem – ela é automaticamente convertida de buffer para string.

Não precisamos vincular o socket a uma porta. Entretanto, sem essa associação, o socket tentaria escutar o tráfego de todas as portas.

Exemplo 7.6 – Um servidor baseado em socket UDP, esperando por mensagens na porta 8124

```
var dgram = require('dgram');

var server = dgram.createSocket("udp4");

server.on ("message", function(msg, rinfo) {
   console.log("Message: " + msg + " from " + rinfo.address + ":"
      + rinfo.port);
});

server.bind(8124);
```

Não precisei chamar o método `close` no cliente nem no servidor ao enviar ou receber mensagens. Não existe uma conexão sendo administrada entre o cliente e o servidor – só o que há são sockets capazes de receber e transmitir dados.

Sentinelas nos portões

A segurança de aplicações web vai além de garantir que as pessoas não acessem o servidor da aplicação. Na verdade, a segurança pode ser um assunto bastante complicado e até mesmo intimidante. Felizmente, quando falamos em aplicações do Node, muitos dos componentes dos quais precisaríamos para isso já foram criados. Basta plugá-los na aplicação, no lugar e momento apropriados.

TLS/SSL

Podemos implementar um canal de comunicação seguro e resistente a bisbilhoteiros empregando uma tecnologia batizada de Secure Sockets Layer (SSL), que recentemente mudou de nome para Transport Layer Security (TLS). O TLS/SSL é a tecnologia que implementa o recurso de criptografia usado no HTTPS, que veremos na próxima seção. Antes de desenvolver em HTTPS, precisamos ajeitar nosso ambiente.

Uma conexão em TLS/SSL requer um *handshake* (em português, aperto de mãos, no sentido de uma apresentação inicial antes de começar a conversa) entre o cliente e o servidor. Durante o handshake, o cliente (tipicamente o navegador) informa ao servidor quais tipos e funções de criptografia são suportados. O servidor escolhe uma das funções e envia um *certificado SSL*, que inclui uma chave pública. O cliente confirma o certificado e gera um número aleatório usando a chave do servidor, que é devolvido a ele. O servidor usa sua chave privada para decifrar o número, que por sua vez é usado para embaralhar a comunicação segura.

Para que tudo isso funcione, é preciso gerar o par de chaves pública e privada e o certificado. Em um sistema de produção, o certificado deve ser assinado por uma *autoridade certificadora* (por exemplo, uma empresa de

registro de domínios pode também emitir certificados), , mas durante o desenvolvimento pode-se usar um *certificado autoassinado*. Este último faz com que o navegador do usuário reclame com veemência sempre que visitar a aplicação, o que não é um problema nessa fase.

Evitando os alarmes de certificado autoassinado

Se estiver usando um certificado autoassinado, é possível contornar os alertas se acessarmos a aplicação em Node pelo localhost (i.e., *https://localhost:8124*). Outra alternativa é usar o site Lets Encrypt, que cria um certificado válido sem termos de pagar o preço praticado pelas entidades certificadoras. O Lets Encrypt encontra-se atualmente em testes, mas já está aberto para o público e dispõe de documentação ensinando a gerar seu próprio certificado.

A ferramenta usada para gerar os arquivos necessários é OpenSSL. No Linux, deve estar instalado por default. Há um instalador para Windows no site oficial. No caso dos produtos Apple, a empresa tem seu próprio ambiente de criptografia. Aqui, veremos apenas a configuração do ambiente Linux.

Para iniciar, digite o comando:

```
openssl genrsa -des3 -out site.key 2048
```

O comando gera uma chave privada, criptografada com Triple-DES e armazenada em formato privacy-enhanced mail (PEM), tornando-a legível em modo texto.

É preciso informar uma senha, que será usada no próximo passo: a criação de uma solicitação de endosso de certificado (certificate-signing request – CSR).

Ao gerar o CSR, o sistema pede a senha que acabamos de criar. Logo em seguida, um número bastante grande de perguntas será apresentado, em inglês, incluindo o código de país (como, por exemplo, US para Estados Unidos), seu estado ou província, nome da cidade, nome da empresa e organização e endereço de email. A pergunta mais importante é a que pede o Common Name. O que isso está solicitando na verdade é o nome de domínio do site ou rede – por exemplo, *burningbird.net* ou *suaempresa.com*. Informe o hostname da máquina em que a aplicação está sendo hospedada. Em meu exemplo, criei um certificado para `examples.burningbird.net`.

```
openssl req -new -key site.key -out site.csr
```

A chave privada requer uma senha grande, ou *passphrase*. O problema é que toda vez que o servidor for reiniciado é necessário informar essa passphrase. Em um sistema em produção, isso é descabido. No próximo passo, removeremos a passphrase de dentro da chave. Primeiro, renomeie a chave:

```
mv site.key site.key.org
```

Depois digite:

```
openssl rsa -in site.key.org -out site.key
```

Ao remover a passphrase, certifique-se de que o servidor está seguro fazendo com que o arquivo esteja legível apenas para usuários e grupos confiáveis.

O próximo passo é gerar o certificado autoassinado. O comando a seguir cria um desses certificados, válido por 365 dias:

```
openssl x509 -req -days 365 -in site.csr -signkey site.key -out final.crt
```

Pronto! Agora temos todos os componentes necessários para usar TLS/SSL e HTTPS.

Trabalhando com HTTPS

Quando uma página na internet pede para o usuário fazer login ou informar dados de cartão de crédito, toda a comunicação deveria acontecer pelo protocolo HTTPS. Se ele não estiver sendo usado, os dados são transmitidos de forma aberta e podem ser facilmente capturados por terceiros mal-intencionados. O HTTPS é uma variante do protocolo HTTP que combina este último com o SSL, garantindo com isso que o site seja o que realmente diz ser, que os dados estejam sendo criptografados antes do envio e que os dados que chegam na outra ponta não tenham sido adulterados.

A inclusão do suporte a HTTPS em uma aplicação Node é semelhante ao que fizemos com o HTTP. A única providência adicional é incluir um objeto `options`, que fornece a chave de criptografia pública e o certificado assinado. A porta default para um servidor HTTPS também é diferente: o

HTTP é servido na porta 80, por default, enquanto o HTTPS normalmente é oferecido na porta 443.

Não precisa indicar a porta!

Uma das vantagens de usar SSL em uma aplicação no Node é que a porta-padrão do HTTPS é 443. Isso significa que não precisamos especificar o número de porta para acessar a aplicação, e também não teremos conflitos com o servidor Apache (ou outro qualquer) que já exista na máquina. Claro, essa vantagem acaba se o Apache também estiver usando HTTPS.

O Exemplo 7.7 demonstra um servidor HTTPS bastante enxuto. Basicamente, é só um pouco mais sofisticado do que nosso programa de Hello World do Capítulo 1.

Exemplo 7.7 – Um servidor HTTPS bastante simples

```
var fs = require("fs"),
    https = require("https");

var privateKey = fs.readFileSync('site.key').toString();
var certificate = fs.readFileSync('final.crt').toString();

var options = {
   key: privateKey,
   cert: certificate
};

https.createServer(options, function(req,res) {
   res.writeHead(200);
   res.end("Hello Secure World\n");
}).listen(443);
```

A chave pública e o certificado são abertos e seus conteúdos são lidos de forma síncrona. Os dados são anexados ao objeto `options`, passados como primeiro parâmetro do método `https.createServer`. A função de callback para o mesmo método é nossa velha conhecida, contendo a solicitação ao servidor e o objeto resposta passados como parâmetros.

Para executar essa aplicação em Node, precisamos do nível de permissão do usuário root, pois o servidor deve estar associado à porta 443. Qualquer serviço que rode em uma porta abaixo de 1024 requer privilégios de root. Podemos rodá-lo usando outra porta, como a 3000, e tudo funcionará às mil maravilhas. A única ressalva é que, se a porta for diferente de 443, é necessário indicar o número de porta no URL:

```
https://examples.burningbird.net:3000
```

Ao acessar a página, vemos o que acontece quando usamos um certificado autoassinado, como mostra a Figura 7.1. Não é nem um pouco difícil entender por que um certificado autoassinado só deva ser usado durante a fase de testes. Acessar a página pelo endereço de loopback (localhost) também evita a mensagem de segurança.
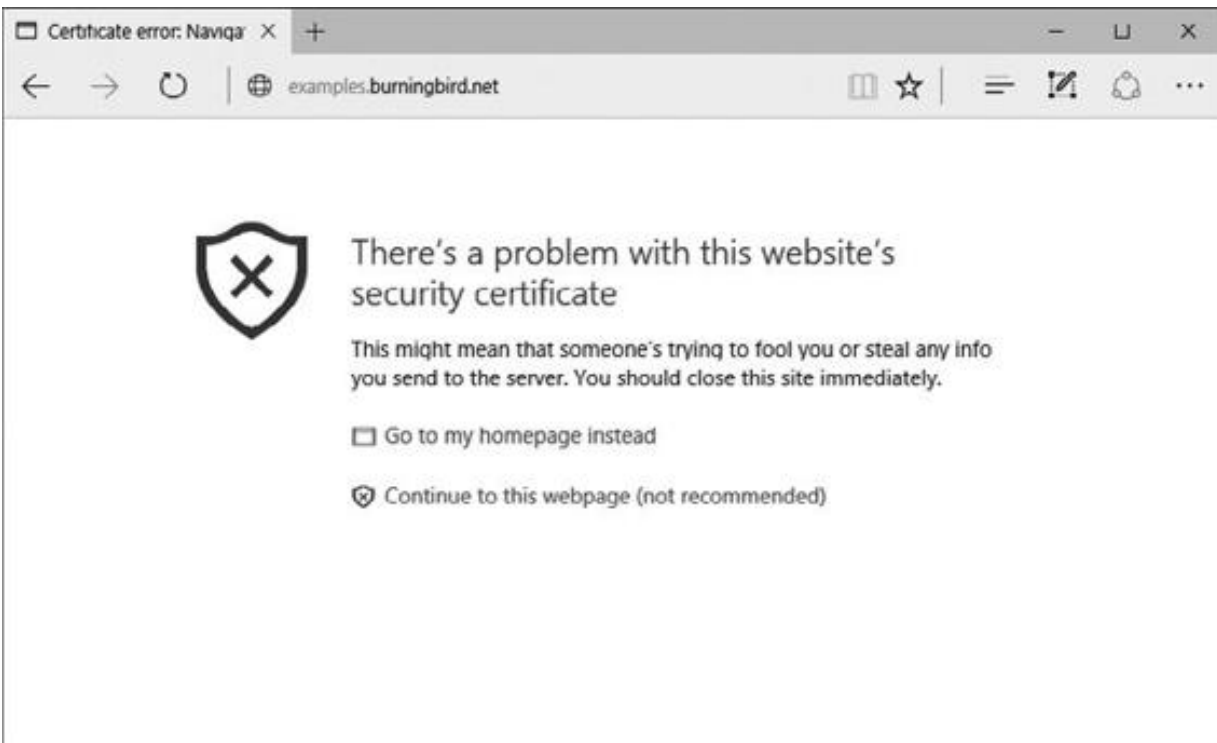
Figura 7.1 – É isso o que acontece quando usamos o Edge para acessar um site em HTTPS com um certificado autoassinado.

A barra de endereços do navegador demonstra outra maneira pela qual o navegador nos indica que o certificado do site não é confiável, como mostrado na Figura 7.2. Em vez de mostrar um cadeado fechado, indicando que o site está numa conexão segura via HTTPS, a barra mostra um cadeado com um belo *x* vermelho, indicando que o certificado não é confiável. Se clicarmos no cadeado, abre-se uma janela com informações sobre o certificado suspeito.

Novamente, se adquirirmos um certificado endossado por uma entidade certificadora, eliminamos todas essas mensagens intimidadoras. Como agora temos opções sem custo para tal certificado, não há mais desculpas

para não usar HTTPS em tudo, não apenas nas aplicações que lidam com senhas ou transações financeiras.
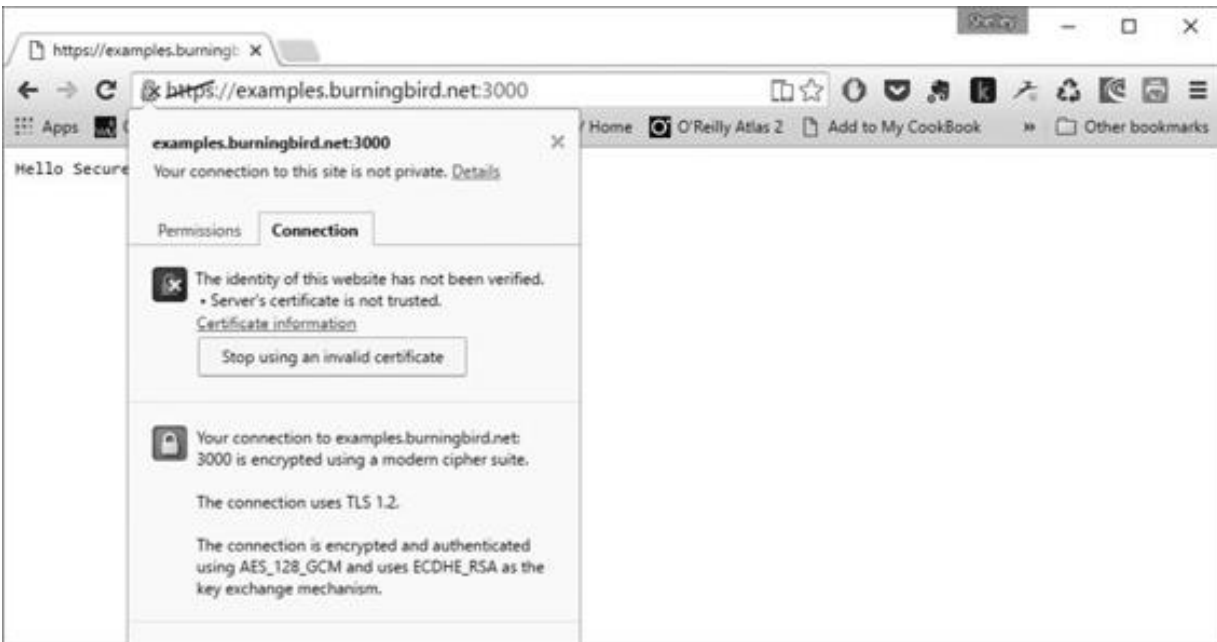
Figura 7.2 – Informações sobre o certificado são mostradas quando clicamos no ícone do cadeado, neste caso no Chrome.

Módulo Crypto

O Node tem um módulo exclusivo para criptografia chamado Crypto, que nada mais é do que uma interface para a funcionalidade do OpenSSL. Isso inclui wrappers para as funções hash, HMAC, cipher, decipher, sign e verify do OpenSSL. O componente do Node que implementa a tecnologia é, na verdade, bastante simples de usar, mas há um consenso bastante forte de que o desenvolvedor de Node precisa conhecer e entender o OpenSSL e todas as suas funções.

Familiarize-se com o OpenSSL

Uma dica quase obrigatória: reserve um tempo para estudar e conhecer muito bem o OpenSSL antes de trabalhar com o módulo Crypto do Node. Recomendo explorar toda a documentação do OpenSSL, bem como ler o e-book gratuito de Ivan Ristić, *OpenSSL Cookbook*.

Estudaremos nas próximas páginas um dos usos mais diretos e simples do módulo Crypto, que é criar um hash das senhas de usuário usando a

funcionalidade `hash` do OpenSSL. A mesma funcionalidade pode ser usada para criar um hash que servirá como *checksum*, garantindo a integridade de dados transmitidos ou armazenados no processo.

MySQL

O exemplo mostrado a seguir emprega o gerenciador de banco de dados MySQL. Para mais informações sobre o módulo node-mysql, consulte seu repositório no GitHub. Se o leitor não tiver acesso ao MySQL, pode armazenar o nome do usuário, senha e salt em um arquivo local e modificar o exemplo para acessá-lo.

Podemos usar o método `createHash` do módulo Crypto para criar um hash de senha antes de armazená-la em um banco de dados. O exemplo a seguir cria um hash pelo algoritmo `sha1`, usa o hash para codificar a senha e extrai o digest dos dados para guardá-los no banco:

```
var hashpassword = crypto.createHash('sha1')
                                  .update(password)
                                  .digest('hex');
```

A codificação do digest está em hexadecimal. A codificação é binária por default, mas também podemos usar base64.

O que é melhor: criptografar a senha ou guardar seu hash?

Guardar uma senha criptografada é obviamente melhor do que armazenar a senha sem criptografia nenhuma, como texto puro. Todavia, as senhas criptografadas podem ser quebradas por uma agência ou um indivíduo que obtenha a chave de criptografia usada para a cifragem. O método mais seguro é não guardar a senha, mas apenas o seu hash, pois um hash normalmente não pode ser decifrado para obter a senha que o gerou. Se uma pessoa perder a senha, não será possível recuperá-la, o sistema permite apenas que ela seja redefinida (i.e., gerar uma nova senha em vez de recuperar a antiga). Para mais informações, consulte *http://www.darkreading.com/safely-storing-user-passwords-hashing-vs-encrypting/a/d-id/1269374*.

Muitas aplicações usam hashes para esse fim. Entretanto, há um problema em armazenar os hashes de senha em um banco de dados, um problema conhecido pela alcunha quase inocente de *rainbow table* (tabela arco-íris).

Explicando da maneira mais simples possível, uma rainbow table é basicamente uma tabela de valores de hash pré-computados para todas as combinações possíveis de caracteres. Portanto, mesmo que tenhamos uma senha forte o suficiente que não possa ser comprometida – e, sejamos honestos, quase nenhum de nós usa senhas fortes –, há grande probabilidade de que alguma sequência de caracteres da senha possa ser

encontrada numa rainbow table, o que torna muito mais simples o trabalho humano de “chutar” a senha correta a partir das pistas encontradas.

A maneira de contornar o problema das rainbow tables é usar tempero, conhecido como *salt* (sal, em português). Um tempero, ou salt, é um valor gerado pelo sistema que é concatenado com a senha antes de criptografá-la. A forma mais simples de implementar o salt é ter um único valor usado em todas as senhas e armazenado de forma segura no servidor. Uma opção mais robusta seria gerar um salt diferente para cada senha e guardá-lo em outro campo do banco de dados. É verdade, o salt pode ser roubado no mesmo momento que a senha, mas seria necessário gerar uma rainbow table específica para cada senha que se queira quebrar, o que praticamente inviabiliza o processo.

O Exemplo 7.8 é uma aplicação simples que recebe o login e a senha passados como argumento de linha de comando, gera o hash e armazena os três dados como um novo usuário numa tabela no MySQL. O exemplo emprega este em vez de outro gerenciador de banco de dados porque o MySQL é praticamente onipresente na maioria dos sistemas e é seguramente o SGBD open source mais popular.

Para prosseguir com o exemplo, instale o módulo node-mysql:

```
npm install node-mysql
```

Crie a tabela com o seguinte comando em SQL:

```
CREATE TABLE user (userid INT NOT NULL AUTO_INCREMENT, PRIMARY KEY(userid),
  username VARCHAR(400) NOT NULL, passwordhash VARCHAR(400) NOT NULL,
  salt DOUBLE NOT NULL );
```

O salt consiste de um valor multiplicado por um número aleatório e arredondado e é concatenado à senha antes que o hash seja criado. Depois disso, todos os dados do usuário são inseridos em uma tabela do MySQL chamada user.

Exemplo 7.8 – O método createHash do módulo Crypto usa o salt para cifrar uma senha

```
var mysql = require('mysql'),
    crypto = require('crypto');
```

```
var connection = mysql.createConnection({
   host: 'localhost',
   user: 'username',
   password: 'userpass'
});

connection.connect();

connection.query('USE nodedatabase');

var username = process.argv[2];
var password = process.argv[3];

var salt = Math.round((Date.now() * Math.random())) + '';

var hashpassword = crypto.createHash('sha512')
                   .update(salt + password, 'utf8')
                   .digest('hex');
// cria a ficha do usuário no banco
connection.query('INSERT INTO user ' +
   'SET username = ?, passwordhash = ?, salt = ?
   [username, hashpassword, salt], function(err, result) {
      if (err) console.error(err);
         connection.end();
});
```

Seguindo o código na ordem, a primeira ação é estabelecer uma conexão com o banco de dados. Depois, o banco e a tabela são selecionados. O login e a senha são extraídos da linha de comando, e a partir daí a mágica do Crypto se revela.

O salt é gerado e passado para a função para criar um hash pelo algoritmo sha512. As funções para atualizar o hash da senha com o salt e para definir a codificação do hash são encadeadas à função que efetivamente cria o hash. Então, o novo hash (e o nome de login) são inseridos na tabela.

A aplicação que testa login e senha está no Exemplo 7.9, que simula um programa cliente que fará login de um usuário já cadastrado. O programa extrai login e senha da linha de comando e depois procura o salt e o hash da senha, existentes no banco de dados, usando o login como chave de pesquisa. O salt é usado para gerar um novo hash baseado na senha fornecida pelo usuário. Se esse novo hash for idêntico ao hash guardado no banco de dados, o usuário digitou a senha correta e pode entrar no

sistema. Se os dois hashes não baterem, o usuário não é validado e tem seu acesso negado.

Exemplo 7.9 – Autenticação de usuário com login e senha

```
var mysql = require('mysql'),
    crypto = require('crypto');

var connection = mysql.createConnection({
   user: 'username',
   password: 'userpass'
});

connection.query('USE nodedatabase');

var username = process.argv[2];
var password = process.argv[3];

connection.query('SELECT password, salt FROM user WHERE username = ?',
   [username], function(err, result, fields) {
   if (err) return console.error(err);

   var newhash = crypto.createHash('sha512')
                 .update(result[0].salt + password, 'utf8')
                 .digest('hex');

if (result[0].password === newhash) {
   console.log("OK, you're cool");
} else {
   console.log("Your password is wrong. Try again.");
}
connection.end();
});
```

Cadastramos o usuário na primeira aplicação, com o nome de login `Michael` e a senha `apple*frk13*`:

```
node password.js Michael apple*frk13*
```

Quando tentamos login com os mesmos usuário e senha:

```
node check.js Michael apple*frk13*
```

temos a seguinte resposta:

```
OK, you're cool
```

Mas se digitarmos uma senha diferente:

```
node check.js Michael badstuff
```

somos recebidos com uma negativa (o que é esperado):

```
Your password is wrong. Try again
```

O hash criptografado pode também ser usado em um fluxo (stream). Por exemplo, digamos que exista um checksum, que é uma maneira algorítmica de determinar se os dados estão íntegros ou não. Criamos um hash do arquivo e o enviamos junto com o arquivo original ao transmiti-lo. A pessoa que baixa o arquivo pode então usar o hash para verificar a precisão da transmissão. O código a seguir usa a função `pipe()` e a natureza duplex das funções do módulo Crypto para criar esse tipo de hash:

```
var crypto = require('crypto');
var fs = require('fs');
var hash = crypto.createHash('sha256');
hash.setEncoding('hex');

var input = fs.createReadStream('main.txt');
var output = fs.createWriteStream('mainhash.txt');

input.pipe(hash).pipe(output);
```

Podemos usar `md5` como algoritmo para gerar o checksum. O MD5 é popular por ser bem rápido, embora já esteja obsoleto e seja bastante inseguro.

```
var hash = crypto.createHash('md5');
```

---

1 N. do T.: Apesar de a autora não indicar aqui, ela está falando apenas de pacotes TCP. Datagramas UDP não tem essa sinalização com as flags SYN, ACK, RST, URG e FIN.

CAPÍTULO 8

Processos-filho

Os sistemas operacionais dão acesso a uma gama muito grande de funcionalidades, mas a maioria delas só pode ser usada pela linha de comando. Seria ótimo se nossas aplicações em Node pudessem acessar todas as funcionalidades. É aí que os *processos-filho* entram.

O Node nos permite executar um comando do sistema operacional dentro de um processo-filho e monitorar seus fluxos de entrada e saída, o que inclui passar argumentos ao comando e até mesmo redirecionar (com pipe) os resultados de um comando para o outro. Nas próximas páginas, exploraremos essa funcionalidade em detalhes.

Todos os exemplos demonstrados neste capítulo, com exceção dos últimos, são comandos do Unix, portanto funcionam no Linux e no OS X. Obviamente, eles não funcionarão no Windows.

child_process.spawn

Existem quatro técnicas diferentes para criar um processo-filho. A mais comum é a que usa o método `spawn`. Essa técnica lança um comando em um novo processo, passando a ele quaisquer argumentos informados. São estabelecidos pipes entre a aplicação-pai e o processo-filho nos fluxos `stdin`, `stdout` e `stderr`.

No exemplo a seguir, criamos um processo-filho para chamar o comando `pwd` do Unix, que mostra o caminho da pasta atual. O comando não recebe nenhum argumento:

```
var spawn = require('child_process').spawn,
    pwd = spawn('pwd');

pwd.stdout.on('data', function (data) {
    console.log('stdout: ' + data);
```

```
});

pwd.stderr.on('data', function (data) {
   console.error('stderr: ' + data);
});

pwd.on('close', function (code) {
   console.log('child process exited with code ' + code);
});
```

Observe que os eventos são capturados a partir dos fluxos `stdout` e `stderr` do processo-filho. Se não ocorrer nenhum erro, a saída do comando é transmitida ao processo-filho `stdout`, disparando um evento `data` no processo. Se algum erro aparecer, como no exemplo a seguir, no qual passamos uma opção inválida ao comando:

```
var spawn = require('child_process').spawn,
    pwd = spawn('pwd', ['-g']);
```

o erro é enviado a `stderr`, que imprime o erro no console:

```
stderr: pwd: invalid option -- 'g'
Try `pwd --help' for more information.

child process exited with code 1
```

O processo é encerrado com o código `1`, que sinaliza a ocorrência de um erro. O código de saída varia, dependendo do sistema operacional ou do erro. Quando nenhum erro ocorre, o processo-filho é encerrado com o código `0`.

O código anterior demonstra o envio dos dados de saída a `stdout` e `stderr` do processo-filho, mas e `stdin`? A documentação do Node para processos-filho inclui um exemplo de redirecionamento de dados para `stdin`. Ele é usado para emular o pipe (|) do Unix, no qual a saída de um comando é redirecionada para a entrada do próximo. Adaptei o exemplo para demonstrar um dos muitos usos do Unix pipe – poder vasculhar todas as pastas, dentro da pasta local, procurando um arquivo com uma palavra específica (neste caso, *test*) em seu nome:

```
find . -ls | grep test
```

O Exemplo 8.1 implementa essa funcionalidade como um processo-filho. Observe que o primeiro comando, que executa o `find`, recebe dois argumentos, enquanto o segundo recebe apenas um: um termo passado

pelo usuário por meio do `stdin`. Observe também que a codificação do `stdout` do processo-filho do `grep` é alterada via `setEncoding`. Se não fosse assim, quando os dados fossem mostrados na tela, o formato seria de buffer.

Exemplo 8.1 – Empregando processos-filho para encontrar arquivos com a palavra "test" em uma estrutura de pastas

```
var spawn = require('child_process').spawn,
    find = spawn('find',['.','-ls']),
    grep = spawn('grep',['test']);

grep.stdout.setEncoding('utf8');

// redireciona com pipe a saída do find para a entrada do grep
find.stdout.pipe(grep.stdin);

// roda o grep e mostra os resultados
grep.stdout.on('data', function (data) {
   console.log(data);
});

// tratamento de erros para ambos
find.stderr.on('data', function (data) {
   console.log('grep stderr: ' + data);
});
grep.stderr.on('data', function (data) {
   console.log('grep stderr: ' + data);
});

// tratamento de saída para ambos
find.on('close', function (code) {
   if (code !== 0) {
      console.log('find process exited with code ' + code);
   }
});

grep.on('exit', function (code) {
   if (code !== 0) {
      console.log('grep process exited with code ' + code);
   }
});
```

Ao executar a aplicação, obtemos uma listagem de todos os arquivos na pasta atual e em qualquer subpasta que contenha a palavra *test* em seu nome.

A documentação do Node alerta que alguns programas usam internamente I/O com buffer de linhas. Com isso, os dados sendo enviados ao programa podem não ser consumidos imediatamente. Para nós, isso significa que, com processos-filho, os dados são armazenados em blocos no buffer antes de serem processados. O processo-filho `grep` é um desses processos.

No exemplo atual, a saída do processo `find` é limitada, portanto a entrada do processo `grep` não excede o tamanho do bloco (tipicamente 4096, mas pode diferir baseado no sistema operacional e em configurações individuais).

O buffer no stdio

Para saber mais sobre buffers, consulte "How to fix stdio buffering" (*https://www.perkin.org.uk/posts/how-to-fix-stdio-buffering.html*) e "Buffering in standard streams" (*http://www.pixelbeat.org/programming/stdio_buffering/*).

Podemos desligar os buffers para o `grep` com a opção `--line-buffered`. Na aplicação a seguir – usando o comando `ps` para examinar os processos em execução e procurar instâncias do apache2 – o buffer de linha foi desligado para o `grep` e os dados são exibidos na tela imediatamente, sem esperar que o buffer encha:

```
var spawn = require('child_process').spawn,
    ps = spawn('ps', ['ax']),
    grep = spawn('grep', ['--line-buffered', 'apache2']);

ps.stdout.pipe(grep.stdin);

ps.stderr.on('data', function (data) {
   console.log('ps stderr: ' + data);
});

ps.on('close', function (code) {
   if (code !== 0) {
      console.log('ps process exited with code ' + code);
   }
});

grep.stdout.on('data', function (data) {
   console.log('' + data);
});

grep.stderr.on('data', function (data) {
```

```
  console.log('grep stderr: ' + data);
});

grep.on('close', function (code) {
  if (code !== 0) {
    console.log('grep process exited with code ' + code);
  }
});
```

Agora, a saída não passa por um buffer e é mostrada imediatamente.

O método `child_process.spawn()` não executa o comando externo em um shell por default. Contudo, a partir do Node 5.7.0 podemos especificar a opção `shell`, para que o processo-filho seja executado em um. Mais adiante, em "Executando uma aplicação com processos-filho no Windows", demonstro como isso funciona em uma aplicação Node destinada ao ambiente Windows. Há também outras opções, incluindo as seguintes (a lista não está completa; verifique a documentação do Node para uma lista abrangente e atualizada):

- `cwd` – Muda a pasta (diretório) atual.
- `env` – Um array com pares chave/valor do ambiente.
- `detached` – Prepara o processo-filho para rodar independente do processo-pai.
- `stdio` – Um array com as opções de `stdio` do processo-filho.

A opção `detached`, como `shell`, mostra diferenças interessantes se compararmos um ambiente Windows com os outros. Veremos mais sobre elas em "Executando uma aplicação com processos-filho no Windows".

A função `child_process.spawnSync()` é a versão síncrona da mesma função.

child_process.exec e child_process.execFile

Além de iniciar um processo-filho, você pode usar `child_process.exec()` e `child_process.execFile()` para executar um comando.

O método `child_process.exec()` é semelhante a `child_process.spawn()` com a exceção de que `spawn()` inicia devolvendo um fluxo tão logo o programa executa, como vimos no Exemplo 8.1. A função `child_process.exec()`, como `child_process.execFile()`, guarda o resultado num buffer. Entretanto, `exec()`

cria um shell para processar a aplicação, diferente de `child_process.execFile()`, que roda o processo diretamente. Isso faz com que `child_process.execFile()` seja mais eficiente do que `child_process.spawn()` com a opção `shell` ou `child_process.exec()`.

O primeiro parâmetro em `child_process.exec()` ou `child_process.execFile()` é o comando (para o `exec()`) ou o arquivo e sua localização (`execFile()`); o segundo parâmetro pode trazer mais opções para o comando. O terceiro é uma função de callback. A função de callback recebe três argumentos: `error`, `stdout` e `stderr`. Os dados são armazenados em buffer antes de serem redirecionados a `stdout` caso nenhum erro ocorra.

Se o arquivo executável contiver:

```
#!/usr/bin/node

console.log(global);
```

a aplicação a seguir mostra no console os resultados guardados nos buffers:

```
var execfile = require('child_process').execFile,
    child;

child = execfile('./app', function(error, stdout, stderr) {
   if (error == null) {
      console.log('stdout: ' + stdout);
   }
});
```

o que também pode ser conseguido com `child_process.exec()`:

```
var exec = require('child_process').exec,
    child;

child = exec('./app', function(error, stdout, stderr) {
   if (error) return console.error(error);
   console.log('stdout: ' + stdout);
});
```

A diferença é que `child_process.exec()` inicia um shell, enquanto `child_process.execFile()` não faz isso.

A função `child_process.exec()` recebe três parâmetros: o comando, um objeto `options` e um callback. O objeto `options` recebe um grande número de valores do processo, incluindo `encoding`, `uid` (user id, o número que

identifica o usuário no Unix) e `gid` (group id, o identificador do grupo). No Capítulo 6, criamos uma aplicação que copia um arquivo PNG e adiciona um efeito Polaroid. Ele usa um processo-filho (spawn) para acessar o ImageMagick, uma ferramenta gráfica poderosa. Para executá-la usando `child_process.exec()`, use o código a seguir, que incorpora um argumento de linha de comando:

```
var exec = require('child_process').exec,
    child;

child = exec('./polaroid -s phoenix5a.png -f phoenixpolaroid.png',
   {cwd: 'snaps'}, function(error, stdout, stderr) {
      if (error) return console.error(error);
         console.log('stdout: ' + stdout);
});
```

`child_process.execFile()` tem um parâmetro adicional: um array de opções de linha de comando que são passadas à aplicação. A aplicação equivalente que usa essa função é:

```
var execfile = require('child_process').execFile,
    child;

child = execfile('./snapshot',
   ['-s', 'phoenix5a.png', '-f', 'phoenixpolaroid.png'],
   {cwd: 'snaps'}, function(error, stdout, stderr) {
      if (error) return console.error(error);
      console.log('stdout: ' + stdout);
});
```

Observe que os argumentos de linha de comando são separados em diferentes elementos de array com um valor para cada argumento.

Como `child_process.execFile()` não inicia um shell, não pode ser usado em algumas circunstâncias. A documentação diz que não podemos usar redirecionamento de I/O nem *file globbing* usando expansões de nome de arquivo (via expressões regulares ou curingas). Contudo, se estiver tentando rodar um processo-filho de forma interativa, use `child_process.execFile()` em vez de `child_process.exec()`. O código a seguir, criado por Colin Ihrig, membro da Node Foundation, demonstra isso com maestria:

```
'use strict';
```

```
const cp = require('child_process');
const child = cp.execFile('node', ['-i'], (err, stdout, stderr) => {
   console.log(stdout);
});

child.stdin.write('process.versions;\n');
child.stdin.end();
```

A aplicação inicia com uma sessão interativa do Node, pergunta pelas versões dos processos e encerra o modo de entrada.

Há versões síncronas – `child_process.execSync()` e `child_process.execFileSync()` – de ambas as funções.

child_process.fork

O último método de criação de processos-filho que veremos é `child_process.fork()`. Essa variação de `spawn()` serve para iniciar processos do Node.

O que distingue `child_process.fork()` dos demais é que há um canal de comunicação estabelecido com o processo-filho. Por outro lado, cada novo processo consome uma instância completa do V8, o que consome memória e tempo de CPU.

Um dos bons usos para `child_process.fork()` é replicar funcionalidade da aplicação para instâncias do Node completamente separadas. Digamos que você tenha uma instância de um servidor Node e queira melhorar o desempenho integrando uma segunda instância do Node para responder a solicitações. A documentação do Node traz um exemplo desses, usando um servidor TCP. Será que podemos usá-lo em paralelo com servidores HTTP? Sim, e a maneira de fazê-lo é semelhante.

Gostaria de agradecer a Jiale Hu por me dar a ideia. Vi certa vez sua demonstração de servidores HTTP paralelos em instâncias separadas. Jiale usa um servidor TCP para passar as terminações dos sockets a dois servidores HTTP filhos separados.

Semelhante ao que foi demonstrado na documentação do Node para servidores TCP paralelos do tipo mestre/filhos, no mestre do meu exemplo, crio o servidor HTTP e então uso a função `child_process.send()` para enviar o servidor ao processo-filho.

```
var cp = require('child_process'),
```

```
    cp1 = cp.fork('child2.js'),
    http = require('http');

var server = http.createServer();

server.on('request', function (req, res) {
   res.writeHead(200, {'Content-Type': 'text/plain'});
   res.end('handled by parent\n');
});

server.on('listening', function () {
   cp1.send('server', server);
});

server.listen(3000);
```

O processo-filho recebe a mensagem do servidor HTTP pelo objeto `process`. Ele escuta por eventos de conexão e, quando recebe, dispara um evento de conexão no servidor HTTP filho, passando a ele o socket que forma o terminador de conexão.

```
var http = require('http');

var server = http.createServer(function (req, res) {
   res.writeHead(200, {'Content-Type': 'text/plain'});
   res.end('handled by child\n');
});

process.on('message', function (msg, httpServer) {
   if (msg === 'server') {
      httpServer.on('connection', function (socket) {
         server.emit('connection', socket);
      });
   }
});
```

Se testarmos a aplicação, acessando o domínio pela porta 3000, veremos que às vezes o servidor HTTP pai trata as solicitações e às vezes é o filho quem trata. Se verificarmos os processos em execução, veremos dois: um para o processo-pai, outro para o filho.

Node Cluster

O módulo Node Cluster é baseado em `child_process.fork()`, além de suas outras funcionalidades.

Executando uma aplicação com processos-filho no Windows

Nas páginas anteriores, aprendemos que os processos-filho que invocam comandos do Unix não funcionariam no Windows, e vice-versa. Pode parecer óbvio, mas nem todo mundo sabe disso e, ao contrário do JavaScript nos navegadores, que é igual em qualquer plataforma, as aplicações em Node podem se comportar de forma diferente em ambientes diferentes.

Em vez de se preocupar com as diferenças entre sistemas, quando estiver no Windows basta usar `child_process.exec()` – que abre um shell para executar a aplicação – ou usar a opção shell das versões mais novas de `child_process.spawn()`. Fora isso, só podemos chamar comandos do Windows usando o interpretador nativo do sistema, `cmd.exe`.

Um exemplo de uso da opção shell com `child_process.spawn()` está na aplicação a seguir, que mostra na tela o conteúdo de uma pasta no Windows:

```
var spawn = require('child_process').spawn,
    pwd = spawn('echo', ['%cd%'], {shell: true});

pwd.stdout.on('data', function (data) {
   console.log('stdout: ' + data);
});

pwd.stderr.on('data', function (data) {
   console.log('stderr: ' + data);
});

pwd.on('close', function (code) {
   console.log('child process exited with code ' + code);
});
```

O comando `echo` envia o resultado do comando `cd` do Windows para a tela. O comando, por sua vez, mostra a pasta atual. Se não tivéssemos ajustado a opção `shell` para `true`, o comando teria falhado.

Um resultado semelhante pode ser obtido com `child_process.exec()`. Entretanto, observe que não precisei usar `echo` com o `child_process.exec()`, pois a saída é guardada em buffer na função seguinte:

```
var exec = require('child_process').exec,
    pwd = exec('cd');

pwd.stdout.on('data', function (data) {
```

```
    console.log('stdout: ' + data);
});
pwd.stderr.on('data', function (data) {
    console.log('stderr: ' + data);
});
pwd.on('close', function (code) {
    console.log('child process exited with code ' + code);
});
```

O Exemplo 8.2 demonstra a terceira opção: rodar comandos do Windows usando o interpretador de comandos do sistema `cmd`, que chama `cmd.exe`. Qualquer argumento após o `cmd` é o que será executado pelo shell do Windows. Na aplicação, o `cmd.exe` é usado para criar uma listagem da pasta, que é ecoada para o console pelo descritor (handler) de eventos de dados.

Exemplo 8.2 – Rodando uma aplicação como um processo-filho em ambiente Windows

```
var cmd = require('child_process').spawn('cmd', ['/c', 'dir\n']);
cmd.stdout.on('data', function (data) {
    console.log('stdout: ' + data);
});
cmd.stderr.on('data', function (data) {
    console.log('stderr: ' + data);
});
cmd.on('exit', function (code) {
    console.log('child process exited with code ' + code);
});
```

A flag `/c` passada como primeiro argumento de `cmd.exe` faz com que este execute o comando e depois encerre a si mesmo. A aplicação não funciona sem essa flag. Por outro lado, você jamais poderá passar a flag `/K`, que instrui `cmd.exe` a executar a aplicação e depois continuar ativo, pois desse modo a aplicação jamais é encerrada.

O equivalente usando `child_process.exec()` é:

```
var cmd = require('child_process').exec('dir');
```

Podemos rodar um cmd ou um arquivo bat usando

`child_process.execFile()`, assim como podemos rodar um arquivo em um ambiente Unix. Considere o seguinte arquivo *my.bat*:

```
@echo off
REM Next command generates a list of program files
dir
```

Execute o arquivo com a seguinte aplicação:

```
var execfile = require('child_process').execFile,
    child;

child = execfile('my.bat', function(error, stdout, stderr) {
   if (error == null) {
      console.log('stdout: ' + stdout);
   }
});
```

CAPÍTULO 9

Node e o ES6

A maioria dos exemplos deste livro usa o JavaScript padrão, que conhecemos há muitos anos. É um código perfeitamente aceitável e familiar às pessoas que programam nele para o navegador. Entretanto, uma das vantagens de desenvolver em Node é que podemos usar versões mais modernas do JavaScript, como o ECMAScript 2015 (ou ES6, como a maioria das pessoas o chama) e posteriores, e não temos de nos preocupar com compatibilidade de navegadores ou sistemas operacionais. Muitos dos novos recursos da linguagem são uma parte inerente da funcionalidade do Node.

Neste capítulo, veremos alguns dos recursos mais novos do JavaScript implementados por default nas versões do Node usadas neste livro. Veremos como aprimorar uma aplicação em Node e as pegadinhas das quais temos de nos precaver quando empregamos uma nova funcionalidade.

Lista de funcionalidades suportadas pelo ES6

Este livro não abrange toda a funcionalidade do ES6 suportada pelo Node – apenas alguns recursos implementados na versão corrente e que já vi usados em aplicações, módulos e exemplos do Node. Para ver uma lista completa dos recursos do ES6, consulte a documentação do Node (*https://nodejs.org/en/docs/es6/*).

Modo strict

O modo strict (rigoroso, em português) do JavaScript existe desde o ECMAScript 5, mas seu uso impacta diretamente a utilização da funcionalidade do ES6, portanto precisamos olhá-lo mais de perto antes de mergulhar no ES6.

O modo strict é ativado quando a linha a seguir é adicionada no início de

uma aplicação em Node:

```
"use strict";
```

Pode-se usar aspas simples ou duplas indiferentemente.

Existem outras maneiras de forçar o modo strict em todos os módulos que são dependências de uma aplicação, como usar a opção `--strict_mode`, mas não recomendo. Forçar o modo strict em um módulo é garantia de que ocorrerão problemas ou consequências imprevistas. Só o use em aplicações ou módulos cujo código tenha sido escrito por você ou esteja sob seu controle.

O modo strict tem impacto significativo no código que você cria. Por exemplo, ele pode gerar erros sempre que uma variável for usada sem antes ser definida, ou sempre que um parâmetro de função for declarado mais de uma vez, ou ainda quando uma variável em uma expressão `eval` estiver no mesmo nível da chamada ao `eval`, entre outros casos. Contudo, a restrição que desejo tratar especificamente aqui é: não podemos usar *literais em formato octal* no modo strict.

Nos capítulos anteriores, quando ajustamos as permissões de um arquivo, aprendemos que não se pode usar octais para definir as permissões:

```
"use strict";

var fs = require('fs');

fs.open('./new.txt','a+',0666, function(err, fd) {
   if (err) return console.error(err);
   fs.write(fd, 'First line', 'utf-8', function(err,written, str) {
      if (err) return console.error(err);
      var buf = new Buffer(written);
      fs.read(fd, buf, 0, written, 0, function (err, bytes, buffer) {
         if (err) return console.error(err);
         console.log(buf.toString('utf8'));
      });
   });
});
```

Em modo strict, este código gera um erro de sintaxe:

```
fs.open('./new.txt','a+',0666, function(err, fd) {
                         ^^^^

SyntaxError: Octal literals are not allowed in strict mode.
```

Podemos converter o octal em um formato seguro se substituirmos o zero à esquerda por um `0o` – um zero seguido da letra “o” minúscula. Ou seja, se usarmos `0o666` em vez de `0666` para definir as permissões, a aplicação funciona:

```
fs.open('./new.txt','a+',0o666, function(err, fd) {
```

A função `fs.open()` também aceita o valor octal como uma string:

```
fs.open('./new.txt','a+','0666', function(err, fd) {
```

Mas essa sintaxe não é muito bem vista pelos mais puristas, então recomendo usar a versão com o `0o`.

O modo strict também é necessário se quisermos usar algumas extensões ES6 – por exemplo, as classes do ES6 só funcionam em modo strict, como veremos mais adiante. A instrução `let`, que discutiremos a seguir, deve ser usada em modo strict.

Saiba mais sobre os octais

Se desejar aprender mais sobre as conversões de literais em octal, recomendo ler uma discussão (em inglês) no fórum ES Discuss intitulada “Octal literals have their uses (you Unix haters skip this one)” (*https://esdiscuss.org/topic/octal-literals-have-their-uses-you-unix-haters-skip-this-one*), cuja tradução aproximada para o português seria “Os octais têm seus usos (se você odeia Unix, nem leia)”.

let e const

Uma limitação que as aplicações em JavaScript tinham no passado era a impossibilidade de declarar variáveis no nível do bloco. Uma das melhores novidades do ES6 é justamente a instrução `let`. Usando-a, podemos declarar uma variável dentro de um bloco, e seu escopo fica restrito a esse bloco. Se usarmos `var`, o valor 100 é mostrado na tela no exemplo a seguir:

```
if (true) {
   var sum = 100;
}

console.log(sum); // exibe 100
```

Porém, se usarmos `let`:

```
"use strict";
```

```
if (true) {
   let sum = 100;
}

console.log(sum);
```

o resultado é completamente diferente:

```
ReferenceError: sum is not defined
```

A aplicação deve estar em modo strict para usar `let`.

Além de confinar o escopo da variável ao bloco, o `let` difere do `var` na elevação de escopo. Variáveis declaradas com `var` são *elevadas* (em inglês, hoisted) ao início do escopo de execução antes que qualquer instrução seja executada. O exemplo a seguir resulta em `undefined` no console, mas não gera nenhum erro durante a execução:

```
console.log(test);
var test;
```

Já este outro exemplo usando `let` resulta em um erro `ReferenceError` durante a execução, informando que `test` não foi definido.

```
"use strict";

console.log(test);
let test;
```

Devemos usar `let` sempre? Alguns programadores dizem que sim, outros juram que não. Podemos usar os dois e limitar o uso do `var` àquelas variáveis que precisam de escopos que abranjam a aplicação toda, ou pelo menos a função toda, e empregar `let` quando o escopo estiver confinado ao bloco de código. Se você é funcionário de alguma empresa, pode ser que ela tenha definido uma norma interna para isso. Caso saiba que seu ambiente o suporta, use `let` e abrace o poder do ES6.

Além do `let`, há outra instrução muito interessante, `const`, que declara um valor de referência apenas de leitura. O valor é uma primitiva, portanto é imutável. Se o valor for um objeto, não se pode inicializar um novo objeto ou primitiva, mas pode-se alterar suas propriedades.

No exemplo a seguir, se tentarmos atribuir um novo valor a uma `const`, a atribuição é ignorada:

```
const MY_VAL = 10;
```

```
MY_VAL = 100;

console.log(MY_VAL); // o console mostra o valor 10
```

É importante observar que `const` é uma referência a um valor. Se atribuirmos um objeto ou array a uma constante definida por `const`, os membros dentro do objeto ou array podem ser alterados, mas não é possível atribuir outro objeto ou array à constante.

```
const test = ['one','two','three'];

const test2 = {apples : 1, peaches: 2};

test = test2; //falha

test[0] = test2.peaches;

test2.apples = test[2];
console.log(test); // [ 2, 'two', 'three' ]
console.log(test2); // { apples: 'three', peaches: 2 }
```

Infelizmente, a instrução `const` já causou bastante confusão. O nome implica uma atribuição *constante* (ou seja, estática), mas uma constante só é imutável para primitivas, nunca para as propriedades internas de um objeto. Se a imutabilidade for importante para o seu código e a constante `const` receber um objeto, podemos empregar o método `object.freeze()` nele para obter ao menos algum tipo de imutabilidade rasa.

Tenho observado que a documentação do Node mostra o uso de `const` ao importar módulos. Ao atribuir um objeto a uma `const`, não conseguimos evitar que suas propriedades sejam modificadas, então o uso de `const` deve ter algum significado semântico que diz a outro programador, em um relance, que não é possível atribuir um novo valor à constante mais tarde no código.

Mais informações sobre constantes e imutabilidade

Mathias Bynens, um dos gurus dos padrões na web (mais conhecidos como web standards), escreveu um artigo técnico muito bom (*https://mathiasbynens.be/notes/es6-const*) sobre `const` e imutabilidade.

Assim como o `let`, o escopo do `const` é restrito ao bloco. Ao contrário do `let`, ele não requer o modo strict.

Podemos usar `let` e `const` em aplicações pela mesma razão que as usaríamos em um navegador, mas nunca encontrei nenhuma vantagem

específica ao Node. Alguns programadores relataram ter obtido um desempenho melhor com `let` e `const`, embora outros tenham reclamado de queda no desempenho. Pessoalmente, jamais notei qualquer mudança, mas sua experiência pode ser diferente da minha. De toda forma, como disse antes, pode ser que sua equipe não tenha de se preocupar com isso, pois a empresa em que trabalha pode ter normatizado o tema e, nesse caso, não há o que escolher.

Funções arrow

Consultando a documentação da API do Node, o aprimoramento mais usado do ES6 são as chamadas *funções arrow*, *funções-flecha* ou ainda (em inglês) *arrow functions*. As funções arrow fazem duas coisas. Primeiro, simplificam a sintaxe. Por exemplo, nos capítulos anteriores usei o trecho de código a seguir para abrir uma nova instância de um servidor HTTP:

```
http.createServer(function (req, res) {

   res.writeHead(200);
   res.write('Hello');
   res.end();

}).listen(8124);
```

Usando uma função arrow (note a seta =>), temos:

```
http.createServer( (req, res) => {

   res.writeHead(200);
   res.write('Hello');
   res.end();

}).listen(8124);
```

A palavra-chave `function` foi removida e a *seta gorda* (`=>`) foi usada para representar a existência de uma função anônima, que recebe os parâmetros dados. A simplificação pode ser ainda maior. Por exemplo, a seguinte estrutura de código de uma função muito familiar para nós:

```
var decArray = [23, 255, 122, 5, 16, 99];
var hexArray = decArray.map(function(element) {
   return element.toString(16);
});

console.log(hexArray); // ["17", "ff", "7a", "5", "10", "63"]
```

pode ser simplificada para:

```
var decArray = [23, 255, 122, 5, 16, 99];
var hexArray = decArray.map(element => element.toString(16));

console.log(hexArray); // [“17”, “ff”, “7a”, “5”, “10”, “63”]
```

As chaves, a instrução `return` e a palavra-chave `function` foram todas removidas, e a mesma funcionalidade foi obtida com um mínimo de código.

As funções arrow não são apenas simplificações de sintaxe, elas também redefinem a palavra-chave `this`. Em JavaScript, antes do aparecimento das funções arrow, cada função definia seu próprio valor para `this`. Assim, no exemplo a seguir, em vez de meu nome ser mostrado no console, obtemos um `undefined`:

```
function NewObj(name) {
   this.name = name;
}

NewObj.prototype.doLater = function() {
   setTimeout(function() {
      console.log(this.name);
   }, 1000);
};

var obj = new NewObj('shelley');
obj.doLater();
```

A razão é simples: `this` foi definido como o objeto no construtor, mas a função `setTimeout` está em uma instância posterior. Contornamos o problema usando outra variável, tipicamente `self`, que pode ser *closed over* – ou seja, fechada, intimamente ligada ao ambiente dado. O código a seguir resulta no comportamento esperado, com meu nome sendo mostrado na tela:

```
function NewObj(name) {
   this.name = name;
}

NewObj.prototype.doLater = function() {
   var self = this;
   setTimeout(function() {
      console.log(self.name);
   }, 1000);
```

```
};

var obj = new NewObj('shelley');
obj.doLater();
```

Em uma função arrow, `this` sempre recebe o valor que normalmente receberia dentro do contexto em que se encontra – neste caso, o objeto `new`:

```
function NewObj(name) {
   this.name = name;
}

NewObj.prototype.doLater = function() {
   setTimeout(()=> {
      console.log(this.name);
   }, 1000);
};

var obj = new NewObj('shelley');
obj.doLater();
```

Contornando as esquisitices das funções arrow

Uma função arrow também tem seus defeitos e idiossincrasias. Por exemplo, como retornar um objeto vazio? Onde estão os argumentos? Strongloop escreveu um artigo excepcional sobre as funções arrow (*https://strongloop.com/strongblog/an-introduction-to-javascript-es6-arrow-functions/*) que discute suas esquisitices e como contorná-las.

Classes

O JavaScript, já há algum tempo, juntou-se a suas irmãs mais velhas no seleto clube das linguagens que suportam classes. Não precisamos mais retorcer seu funcionamento para emular o comportamento que se espera de uma classe em qualquer linguagem.

No Capítulo 3, criamos uma "classe" chamada `InputChecker` usando a sintaxe mais antiga:

```
var util = require('util');
var eventEmitter = require('events').EventEmitter;
var fs = require('fs');

exports.InputChecker = InputChecker;

function InputChecker(name, file) {
   this.name = name;
```

```
   this.writeStream = fs.createWriteStream('./' + file + '.txt',
                     {'flags' : 'a',
                      'encoding' : 'utf8',
                      'mode' : 0666});
};
util.inherits(InputChecker,eventEmitter);
InputChecker.prototype.check = function check(input) {
   var command = input.toString().trim().substr(0,3);
   if (command == 'wr:') {
      this.emit('write',input.substr(3,input.length));
   } else if (command == 'en:') {
      this.emit('end');
   } else {
      this.emit('echo',input);
   }
};
```

Modifiquei-a levemente para embutir a função `check()` diretamente na definição do objeto. Depois, converti o resultado para uma classe ES6 em modo strict, obrigatório para podermos usar as novas funcionalidades de classe.

Não temos mais de usar `util.inherits()` para que o construtor herde os protótipos dos métodos do superconstrutor. A nova classe estende os métodos do objeto `EventEmitter`. Adicionei o método do construtor para criar e inicializar o novo objeto. Nesse construtor, chamei `super()` para invocar as funções da classe-mãe.

```
'use strict';
const util = require('util');
const eventEmitter = require('events').EventEmitter;
const fs = require('fs');
class InputChecker extends eventEmitter {
   constructor(name, file) {
      super()
      this.name = name;
      this.writeStream = fs.createWriteStream('./' + file + '.txt',
         {'flags' : 'a',
          'encoding' : 'utf8',
          'mode' : 0o666});
      }
```

```
   check (input) {
      let command = input.toString().trim().substr(0,3);
      if (command == 'wr:') {
         this.emit('write',input.substr(3,input.length));
      } else if (command == 'en:') {
         this.emit('end');
      } else {
         this.emit('echo',input);
      }
   }
};

exports.InputChecker = InputChecker;
```

O método `check()` não foi adicionado usando o objeto do protótipo, mas simplesmente incluído na classe como um método. Não há utilidade de definir variáveis ou funções em `check()` – precisamos apenas definir o método. Entretanto, toda a lógica de funcionamento é a mesma ou muito parecida com a do objeto original.

A aplicação que usa o novo módulo "classificado" (perdoe-me, sou péssimo com trocadilhos) é exatamente a mesma:

```
var InputChecker = require('./class').InputChecker;

// testando o novo objeto e a manipulação de eventos
var ic = new InputChecker('Shelley','output');

ic.on('write', function (data) {
   this.writeStream.write(data, 'utf8');
});
ic.addListener('echo', function( data) {
   console.log(this.name + ' wrote ' + data);
});

ic.on('end', function() {
   process.exit();
});

process.stdin.setEncoding('utf8');
process.stdin.on('data', function(input) {
   ic.check(input);
});
```

Como demonstra o código, não precisei colocar a aplicação em modo strict para usar um módulo que esteja em modo strict.

A (sempre útil) documentação Mozilla

A Mozilla Developer Network é sempre primeira parada para consultar qualquer assunto ou detalhe a respeito do JavaScript, e a nova funcionalidade de classes não é exceção. A documentação de referência (*https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes*) sobre o tema é muito bem elaborada.

As promessas de um pássaro azul

Durante os estágios iniciais do desenvolvimento do Node, os criadores debateram bastante sobre o caminho a tomar: callbacks (funções de retorno) ou promises (promessas de função). Os callbacks ganharam, o que fez com que alguns festejassem em júbilo, mas trouxe desapontamento a outros.

As promises são, hoje, parte do ES6, e podemos perfeitamente usá-las em aplicações Node. Entretanto, para que isso seja possível, é preciso implementar o suporte a elas na funcionalidade central do Node, o que pode ser feito modificando o próprio Node ou usando um módulo. Embora tenha tentado passar longe dos módulos de terceiros neste livro, neste caso recomendo e assino embaixo um dos mais populares módulos de terceiros que implementam promises, o chamado Bluebird (pássaro azul).

O Bluebird melhora o desempenho

Outra boa razão para usar o Bluebird é o desempenho. O autor do módulo explica o porquê em um artigo no site StackExchange (*http://softwareengineering.stackexchange.com/questions/278778/why-are-native-es6-promises-slower-and-more-memory-intensive-than-bluebird*).

Em vez de callbacks aninhados, as promises do ES6 permitem subdivisões (branching), de forma que operações executadas com sucesso são tratadas por um dos caminhos da bifurcação e as com falha pelo outro caminho. A melhor maneira de demonstrar esse comportamento é *promisificar* uma aplicação típica de sistema de arquivos no Node – convertendo a estrutura de callbacks em promises.

Com os callbacks nativos, a aplicação a seguir abre um arquivo e lê seu conteúdo, faz uma modificação e depois grava o resultado em outro arquivo.

```
var fs = require('fs');
fs.readFile('./apples.txt','utf8', function(err,data) {
   if (err) {
      console.error(err.stack);
   } else {
      var adjData = data.replace(/apple/g,'orange');
      fs.writeFile('./oranges.txt', adjData, function(err) {
         if (err) console.error(err);
      });
   }
});
```

Mesmo um exemplo simples como esse traz dois níveis de aninhamento: ler o arquivo e depois modificar o conteúdo.

Agora, usaremos o Bluebird para "promisificar" o código.

No exemplo, a função `promisifyAll()` do Bluebird é usada para "promisificar" todas as funções do módulo File System. Em vez de `readFile()`, usaremos `readFileAsync()`, que é a versão dessa função que suporta promises.

```
var promise = require('bluebird');
var fs = promise.promisifyAll(require('fs'));

fs.readFileAsync('./apples.txt','utf8')
   .then(function(data) {
      var adjData = data.replace(/apple/g, 'orange');
      return fs.writeFileAsync('./oranges.txt', adjData);
   })
   .catch(function(error) {
      console.error(error);
   });
```

No exemplo, quando o conteúdo do arquivo é lido, uma operação com sucesso é tratada com a função `then()`. Se a operação falhar, a função `catch()` trata o erro. Se a leitura tiver sucesso, os dados são modificados e a versão promisificada de `writeFile()`, `writeFileAsync()`, é chamada, gravando os dados no arquivo. Olhando o exemplo anterior, com os callbacks aninhados, sabemos que `writeFile()` simplesmente retorna um erro. Esse erro também é tratado pela função `catch()`, bem mais simples.

Mesmo que o exemplo aninhado não seja grande, podemos ver

claramente que o código usando promises é muito mais limpo. Começa a se desenhar a solução para o problema de aninhamento – especialmente porque há apenas uma função de tratamento de erros para qualquer número de chamadas.

Que tal um exemplo mais complexo? Modifiquei o código anterior para incluir um passo adicional: criar uma pasta que contenha o arquivo *oranges.txt*. No código, podemos ver que existem agora duas funções `then()`. A primeira processa a resposta positiva e cria uma pasta, e a segunda cria um arquivo com os dados modificados. O novo diretório é criado usando a função promisificada `mkdirAsync()`, que é devolvida no final do processo. Essa é a chave para que as promises funcionem, pois o próximo `then()` está na verdade ligado à função devolvida. Os dados modificados ainda são repassados à função promise na qual os dados serão gravados. Quaisquer erros na leitura do arquivo ou no processo de criação da pasta são tratados pelo mesmo `catch()`.

```
var promise = require('bluebird');
var fs = promise.promisifyAll(require('fs'));

fs.readFileAsync('./apples.txt','utf8')
   .then(function(data) {
      var adjData = data.replace(/apple/g, 'orange');
      return fs.mkdirAsync('./fruit/');
   })
   .then(function(adjData) {
      return fs.writeFileAsync('./fruit/oranges.txt', adjData);
   })
   .catch(function(error) {
      console.error(error);
   });
```

Então, que tal tratar instâncias em que é devolvido um array de resultados? Um exemplo é a função `readdir()` no módulo File System para ler o conteúdo de uma pasta.

Nessa situação, mostram-se muito úteis as funções de tratamento de arrays como `map()`. No código a seguir, o conteúdo de uma pasta é devolvido como array, e sempre que um arquivo na pasta é aberto, seu conteúdo é modificado e gravado em um arquivo de mesmo nome em

outra pasta. O `catch()` mais interno trata os erros de leitura e escrita dos arquivos, enquanto o mais externo trata o acesso à pasta.

```
var promise = require('bluebird');
var fs = promise.promisifyAll(require('fs'));

fs.readdirAsync('./apples/').map(filename => {
   fs.readFileAsync('./apples/'+filename,'utf8')
      .then(function(data) {
         var adjData = data.replace(/apple/g, 'orange');
         return fs.writeFileAsync('./oranges/'+filename, adjData);
      })
      .catch(function(error) {
         console.error(error);
      })
   })
   .catch(function(error) {
      console.error(error);
   })
```

Neste capítulo, abordei apenas superficialmente o que o Bluebird é capaz de fazer e como é atraente usar promises em Node. Dedique algum tempo para explorar o uso de ambos, bem como de outros recursos novos do ES6, em suas aplicações Node.

CAPÍTULO 10

Desenvolvimento full-stack com Node

A maior parte deste livro tem como foco os módulos e as funcionalidades essenciais que compõem o Node. Procurei evitar a discussão de módulos de terceiros porque o Node ainda é um ambiente muito dinâmico e o suporte a esses módulos pode mudar de forma rápida e drástica.

Todavia, não acho que seja possível discutir o Node sem ao menos mencionar rapidamente o contexto mais amplo das aplicações Node, o que significa que você deverá ter familiaridade com o desenvolvimento full-stack (pilha completa) com o Node. Isso implica ter familiaridade com sistemas de dados, APIs, desenvolvimento do lado cliente – toda uma gama de tecnologias com apenas um ponto em comum: o Node.

A forma mais comum de desenvolvimento full-stack com o Node é a pilha MEAN – MongoDB, Express, AngularJS e Node. No entanto, um desenvolvimento full-stack pode englobar outras ferramentas, como MySQL ou Redis para desenvolvimento de banco de dados, e outros frameworks do lado cliente além do AngularJS. O uso do Express, porém, tem se tornado muito comum. Você deverá ter familiaridade com ele caso queira trabalhar com o Node.

Explorando melhor a pilha MEAN

Para explorações adicionais da pilha MEAN, do desenvolvimento full-stack e do Express, recomendo o livro *Web Development with Node and Express: Leveraging the JavaScript Stack* (O'Reilly, 2014,) de Ethan Brown, *Desenvolvendo com AngularJS* (Novatec, 2014, *https://www.novatec.com.br/livros/angularjs/)* de Shyam Seshadri e Brad Green e o vídeo *Architecture of the MEAN Stack* (O'Reilly, 2015) de Scott Davis.

Framework de aplicação Express

No Capítulo 5, descrevi um pequeno subconjunto das funcionalidades

necessárias para implementar uma aplicação web com Node. A tarefa de criar uma aplicação web com Node, no melhor caso, é intimidadora. É por isso que um framework de aplicação como o Express se tornou tão popular: ele oferece a maioria das funcionalidades com um mínimo de esforço.

O Express está presente em quase todos os lugares no mundo do Node, portanto você deverá ter familiaridade com ele. Veremos a aplicação Express mais simples possível neste capítulo, mas você precisará de um treinamento adicional depois disso.

O Express agora faz parte da Fundação Node.js.

O Express (*http://expressjs.com/*) teve um início problemático, mas agora faz parte da Fundação Node.js (Node.js Foundation). Futuros desenvolvimentos deverão ser mais consistentes e seu suporte deverá ser mais confiável.

O Express oferece uma boa documentação, incluindo uma explicação sobre como iniciar uma aplicação. Seguiremos os passos descritos na documentação e então expandiremos a aplicação básica. Para começar, crie um novo subdiretório para a aplicação e dê-lhe o nome que você desejar. Utilize o npm para criar um arquivo *package.json*, usando *app.js* como o ponto de entrada da aplicação. Por fim, instale o Express, salvando-o em suas dependências no arquivo *package.json*, usando o comando a seguir:

```
npm install express --save
```

A documentação do Express contém uma aplicação Express mínima do tipo Hello World, contida no arquivo *app.js*:

```
var express = require('express');
var app = express();

app.get('/', function (req, res) {
  res.send('Hello World!');
});

app.listen(3000, function () {
  console.log('Example app listening on port 3000!');
});
```

A função `app.get()` trata todas as requisições web do tipo GET, passando os objetos referentes à requisição e à resposta com os quais temos

familiaridade por causa de nossos trabalhos nos capítulos anteriores. Por convenção, as aplicações Express usam as formas abreviadas `req` e `res`. Esses objetos têm a mesma funcionalidade dos objetos default de requisição e resposta, com o acréscimo de novas funcionalidades oferecidas pelo Express. Por exemplo, podemos usar `res.write()` e `res.end()` para responder à requisição web, usados nos capítulos anteriores, mas também podemos usar `res.send()` – uma melhoria do Express – para fazer o mesmo em uma só linha.

Em vez de criar manualmente a aplicação, podemos utilizar também o gerador de aplicações Express para criar o esqueleto da aplicação. Usaremos esse método a seguir, pois ele resulta em uma aplicação Express mais detalhada e completa.

Inicialmente, instale o gerador de aplicações Express globalmente:

```
sudo npm install express-generator -g
```

Em seguida, execute o gerador com o nome que você deseja dar à sua aplicação. Como demonstração, usarei `bookapp`:

```
express bookapp
```

O gerador de aplicações Express cria os subdiretórios necessários. Vá para o subdiretório *bookapp* e instale as dependências:

```
npm install
```

Pronto: você criou o esqueleto de sua primeira aplicação Express. Ela pode ser executada com o comando a seguir se você estiver em um ambiente OS X ou Linux:

```
DEBUG=bookapp:* npm start
```

Execute o seguinte comando se você estiver em uma janela de Comando (Command) do Windows:

```
set DEBUG=bookapp:* & npm start
```

Podemos também iniciar a aplicação somente com `npm start`, deixando de lado a depuração.

A aplicação é iniciada e ouve requisições na porta default do Express, que é a porta 3000. O acesso à aplicação por meio da web faz uma página web simples ser devolvida com a saudação "Welcome to Express" (Bem-vindo

ao Express).

Vários subdiretórios e arquivos são gerados pela aplicação:

```
├── app.js
├── bin
│ └── www
├── package.json
├── public
│ ├── images
│ ├── javascripts
│ └── stylesheets
│ └── style.css
├── routes
│ ├── index.js
│ └── users.js
└── views
    ├── error.jade
    ├── index.jade
    └── layout.jade
```

Veremos mais detalhes sobre muitos dos componentes, mas os arquivos estáticos voltados ao público estão localizados no subdiretório *public*. Conforme você perceberá, os arquivos de imagens e CSS estão nesse local. Os arquivos de template de conteúdo dinâmico estão em *views*. O subdiretório *routes* contém as aplicações de endpoint que ouvem as requisições web e renderizam as páginas.

O Jade agora se chama Pug

Devido à violação de marca registrada, os criadores do Jade não podem mais usar "Jade" como o nome da engine de templates usada pelo Express e por outras aplicações. No entanto, o processo de converter o Jade em Pug ainda está em andamento. Na época em que este livro foi publicado, o Express Generator continuava gerando Jade, mas uma tentativa de instalá-lo como dependência gerava o erro a seguir:

```
Jade has been renamed to pug, please install the latest
version of pug instead of jade
```

O site do Pug (*https://github.com/pugjs*) ainda se chamava Jade, mas a documentação e as funcionalidades permaneciam iguais. Esperamos que tudo isso seja resolvido, antes cedo do que tarde.

O arquivo *www* no subdiretório *bin* é um script de início da aplicação. É um arquivo Node convertido em uma aplicação de linha de comando. Ao observar o arquivo *package.json* gerado, você o verá listado como o script

de início da aplicação.

```
{
  "name": "bookapp",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "start": "node ./bin/www"
  },
  "dependencies": {
    "body-parser": "~1.13.2",
    "cookie-parser": "~1.3.5",
    "debug": "~2.2.0",
    "express": "~4.13.1",
    "jade": "~1.11.0",
    "morgan": "~1.6.1",
    "serve-favicon": "~2.3.0"
  }
}
```

Instale outros scripts para testar, reiniciar ou ainda controlar a sua aplicação no subdiretório *bin*.

Para começar a ter uma visão mais detalhada da aplicação, analisaremos o seu ponto de entrada, que está no arquivo *app.js*.

Ao abrir o arquivo *app.js*, você verá uma quantidade consideravelmente maior de código se comparado à aplicação simples que vimos antes. Vários outros módulos são importados, a maioria dos quais oferece suporte de *middleware*, que seria esperado de uma aplicação voltada à web. Os módulos importados também podem ser específicos da aplicação e estão no subdiretório *routes*:

```
var express = require('express');
var path = require('path');
var favicon = require('serve-favicon');
var logger = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');

var routes = require('./routes/index');
var users = require('./routes/users');

var app = express();
```

Eis os módulos e os seus propósitos:

`express`
: É a aplicação Express.

`path`
: É um módulo nuclear do Node para trabalhar com paths de arquivo.

`serve-favicon`
: É o middleware para servir o arquivo *favicon.ico* a partir de um dado path ou buffer.

`morgon`
: É um logger de requisições HTTP.

`cookie-parser`
: Faz parse do cabeçalho dos cookies e preenche `req.cookies`.

`body-parser`
: Oferece quatro tipos diferentes de parsers para o corpo das requisições (mas não trata corpos com múltiplas partes).

Cada um dos módulos de middleware trabalha com um servidor HTTP básico, bem como com o Express.

O que é middleware?

O middleware é o intermediário entre o sistema/sistema operacional/banco de dados e a aplicação. No Express, o middleware faz parte de uma cadeia de aplicações, em que cada uma executa determinada função relacionada a uma requisição HTTP – seja processá-la ou fazer alguma manipulação na requisição para futuras aplicações do middleware. O conjunto de middlewares que trabalha com o Express (*http://expressjs.com/en/resources/middleware.html*) é bem abrangente.

Na próxima seção de código de *app.js*, o middleware é montado (disponibilizando-o para a aplicação) em um dado path usando a função `app.use()`. A ordem em que o middleware é montado é importante, portanto, se você adicionar novas funcionalidades de middleware, certifique-se de que isso seja feito considerando os outros middlewares, de acordo com as recomendações dos desenvolvedores.

O trecho de código a seguir também inclui código para definir a configuração da engine de visão, que será explicada em breve.

```
// configuração da engine de visão
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');

// remova o caractere de comentário depois de colocar o seu favicon em /public
//app.use(favicon(path.join(__dirname, 'public', 'favicon.ico')));
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));
```

A última chamada a `app.use()` faz referência a um dos poucos middlewares embutidos do Express, `express.static`, usado para tratar todos os arquivos estáticos. Se um usuário web requisitar um arquivo HTML, JPEG ou outro arquivo estático, `express.static` processará a requisição. Todos os arquivos estáticos são servidos em relação ao path especificado quando o middleware é montado – nesse caso, no subdiretório `public`.

Retornando às chamadas da função `app.set()` que definem a engine de visões no trecho de código anterior, você usará uma *engine de templates que ajudará a mapear os dados à entrega*. Um dos mais populares, o Jade, está integrado por padrão, mas outros como o Mustache e o EJS podem ser usados de modo igualmente fácil. A configuração da engine define o subdiretório em que os arquivos de template estão localizados (*views*) e a engine de visão a ser usada (Jade).

Lembrete: o Jade agora se chama Pug

Conforme mencionamos antes neste capítulo, o Jade atualmente se chama Pug (*https://github.com/pugjs*). Consulte tanto a documentação do Express quanto a do Pug para atualizar os exemplos, de modo que eles funcionem com a engine de templates recém-nomeada.

Enquanto este livro estava sendo produzido, modifiquei o arquivo *package.json* gerado a fim de substituir o módulo Jade por Pug:

```
<p>"pug": "2.0.0-alpha8",</p>
```

Além disso, no arquivo `app.js`, substituí a referência a `jade` por `pug`:

```
app.set('view engine', 'pug');
```

A aplicação pareceu funcionar sem problemas.

No subdiretório *views*, você verá três arquivos: *error.jade*, *index.jade* e *layout.jade*. Isso permitirá que você comece a trabalhar, embora vá precisar de muitos outros arquivos quando passar a integrar dados à sua aplicação. A seguir, podemos ver o conteúdo do arquivo *index.jade* gerado:

```
extends layout

block content
  h1= title
  p Welcome to #{title}
```

A linha em que se lê `extends layout` incorpora a sintaxe do Jade contida no arquivo *layout.jade*. Você reconhecerá os elementos referentes ao cabeçalho HTML (`h1`) e a parágrafos (`p`). O cabeçalho `h1` recebe o valor passado para o template como `title`, que também é usado no elemento de parágrafo. O modo como esses valores são renderizados no template exige que retornemos ao arquivo *app.js* para ver o próximo trecho de código:

```
app.use('/', routes);
app.use('/users', users);
```

Esses endpoints são específicos da aplicação e correspondem às funcionalidades para responder às requisições do cliente. A requisição de nível mais alto (`'/'`) é satisfeita pelo arquivo *index.js* no subdiretório *routes*, e `users`, pelo arquivo *users.js*.

No arquivo *index.js*, somos apresentados ao *roteador* (router) do Express, que oferece a funcionalidade de tratamento de respostas. Conforme descrito na documentação do Express, o comportamento do roteador apresenta o seguinte padrão:

```
app.METHOD(PATH, HANDLER)
```

O método é o HTTP, e o Express tem suporte para vários, incluindo os conhecidos `get`, `post`, `put` e `delete`, assim como os possivelmente menos familiares `merge`, `search`, `head`, `options`, e assim por diante. Esse path é o path web, e o handler é a função que processa a requisição. Em *index.js*, o método é `get`, o path é a raiz da aplicação e o handler é uma função de callback que passa a requisição e a resposta:

```
var express = require('express');
var router = express.Router();
```

```
/* GET da página inicial. */
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express' });
});
module.exports = router;
```

Os dados (variáveis locais) e a visão se encontram na chamada da função `res.render()`. A visão usada é o arquivo *index.jade* que vimos antes, e você pode notar que o valor do atributo `title` no template é passado como dado para a função. Em sua cópia, experimente alterar "Express" para o valor que você quiser e recarregue a página para ver a modificação.

O restante do arquivo *app.js* contém tratamento de erros, e deixarei que você explore esse código por conta própria. Essa é uma introdução rápida e bem resumida ao Express, mas espero que, mesmo com esse exemplo simples, você comece a ter uma noção da estrutura de uma aplicação Express.

Incorporando dados

Vou fazer propaganda e recomendar o meu livro *JavaScript Cookbook* (O'Reilly, 2010) caso você queira saber mais sobre incorporação de dados em uma aplicação Express. O Capítulo 14 mostra como estender uma aplicação Express a fim de incorporar um repositório MongoDB, bem como incluir o uso de controladores em uma arquitetura MVC (Model-View-Controller, ou Modelo-Visão-Controlador) completa.

Sistemas de banco de dados MongoDB e Redis

No Capítulo 7, o Exemplo 7.8 mostrou uma aplicação que inseria dados em um banco de dados MySQL. Embora fosse inicialmente rudimentar, o suporte a bancos de dados relacionais em aplicações Node melhorou, com soluções robustas como o driver de MySQL para Node (*https://github.com/mysqljs/mysql*) e novos módulos como o pacote Tedious (*https://github.com/tediousjs/tedious*) para acesso a SQL Server em um ambiente Microsoft Azure.

As aplicações Node também têm acesso a vários outros sistemas de banco de dados. Nesta seção, descreverei rapidamente dois deles: o MongoDB, que é muito popular no desenvolvimento com Node, e o Redis, que é um dos meus favoritos.

MongoDB

O banco de dados mais popular usado em aplicações Node é o MongoDB. Ele é um banco de dados baseado em documentos. Os documentos são codificados como BSON – um formato binário do JSON –, o que provavelmente explica a sua popularidade entre os desenvolvedores de JavaScript. Com o MongoDB, em vez de uma linha de tabela, você terá um documento BSON; em vez de uma tabela, você terá uma coleção.

O MongoDB não é o único banco de dados centrado em documentos. Outras versões populares desse tipo de repositório de dados são o CouchDB da Apache, o SimpleDB da Amazon, o RavenDB e até mesmo o lendário Lotus Notes. Há suporte do Node em graus variados para a maioria dos repositórios modernos de documentos, mas o MongoDB e o CouchDB têm a maior parte.

O MongoDB não é um sistema de banco de dados trivial, e você deverá investir tempo para conhecer suas funcionalidades antes de incorporá-lo em suas aplicações Node. Quando estiver pronto, porém, você encontrará um suporte excelente para o MongoDB no Node por meio do MongoDB Native NodeJS Driver (*https://github.com/mongodb/node-mongodb-native*) e um suporte adicional, orientado a objetos, com o Mongoose (*http://mongoosejs.com/*).

Embora eu não vá descrever os detalhes de como usar o MongoDB com o Node, apresentarei um exemplo, somente para que você possa ter uma ideia de como ele funciona. Apesar de a estrutura de dados subjacente ser diferente daquela dos bancos de dados relacionais, os conceitos permanecem iguais: você criará um banco de dados, criará coleções de registros e adicionará registros individuais. Será possível então atualizar, consultar ou apagar registros. No exemplo com MongoDB mostrado no Exemplo 10.1, vou me conectar com um banco de dados de exemplo, acessar uma coleção de widgets, remover todos os registros existentes, inserir dois e, em seguida, consultar ambos e exibi-los.

Exemplo 10.1 – Trabalhando com um banco de dados MongoDB

```
var MongoClient = require('mongodb').MongoClient;
```

```
// Conecta-se com o banco de dados
MongoClient.connect("mongodb://localhost:27017/exampleDb",
                                        function(err, db) {
  if(err) { return console.error(err); }

  // acessa ou cria a coleção widgets
  db.collection('widgets', function(err, collection) {
    if (err) return console.error(err);

    // remove todos os documentos de widgets
    collection.remove(null,{safe : true}, function(err, result) {
      if (err) return console.error(err);
      console.log('result of remove ' + result.result);

      // cria dois registros
      var widget1 = {title : 'First Great widget',
                     desc : 'greatest widget of all',
                     price : 14.99};
      var widget2 = {title : 'Second Great widget',
                     desc : 'second greatest widget of all',
                     price : 29.99};

      collection.insertOne(widget1, {w:1}, function (err, result) {
       if (err) return console.error(err);
       console.log(result.insertedId);

       collection.insertOne(widget2, {w:1}, function(err, result) {
       if (err) return console.error(err);
       console.log(result.insertedId);

       collection.find({}).toArray(function(err,docs) {
            console.log('found documents');
            console.dir(docs);

            //fecha o banco de dados
            db.close();
          });
        });
      });
    });
  });
});
```

Sim, é um inferno de callbacks do Node, mas você poderá adotar o uso de promises.

O MongoClient é o objeto que você usará com mais frequência para se conectar com o banco de dados. Observe o número da porta especificada

(27017). Essa é a porta default do sistema MongoDB. O banco de dados é `exampleDB`, informado como parte do URL de conexão, e a coleção é `widgets`, por causa da classe Widget factory, amplamente conhecida entre os desenvolvedores.

As funções do MongoDB são assíncronas, conforme esperado. Antes de os registros serem inseridos, a aplicação apaga todos os registros existentes da coleção usando a função `collection.remove()` sem nenhuma query específica. Se não fizéssemos isso, teríamos registros duplicados, pois o MongoDB atribui identificadores únicos gerados pelo sistema a cada novo registro, e não estamos fazendo com que o título nem outro campo sejam explicitamente um identificador único.

Adicionamos cada novo registro com `collection.insertOne()`, passando o JSON que define o objeto. A opção `{w:1}` especifica *write concern* (preocupação com escrita), que descreve o nível de reconhecimento do MongoDB para a operação de escrita.

Depois que os registros são inseridos, a aplicação utiliza `collection.find()`, novamente sem nenhuma query específica, para encontrar todos os registros. Na verdade, a função cria um *cursor*, e a função `toArray()` devolve os seus resultados na forma de um array, que podemos então exibir no console usando `console.dir()`. O resultado da aplicação tem uma aparência semelhante a esta:

```
result of remove 1
56c5f535c51f1b8d712b6552
56c5f535c51f1b8d712b6553
found documents
[ { _id: ObjectID { _bsontype: 'ObjectID', id: 'V..5. \u001f\u001bq+eR' },
    title: 'First Great widget',
    desc: 'greatest widget of all',
    price: 14.99 },
  { _id: ObjectID { _bsontype: 'ObjectID', id: 'V..5.\u001f\u001bq+eS' },
    title: 'Second Great widget',
    desc: 'second greatest widget of all',
    price: 29.99 } ]
```

O identificador do objeto na verdade é um objeto, e o identificador está em BSON, motivo pelo qual ele não é exibido de forma clara. Se quiser ver

uma saída mais limpa, você poderá acessar cada campo individualmente e converter o identificador BSON em uma string hexadecimal com `toHexString()`:

```
docs.forEach(function(doc) {
                    console.log('ID : ' + doc._id.toHexString());
                    console.log('desc : ' + doc.desc);
                    console.log('title : ' + doc.title);
                    console.log('price : ' + doc.price);
                });
```

Eis agora o resultado:

```
result of remove 1
56c5fa40d36a4e7b72bfbef2
56c5fa40d36a4e7b72bfbef3
found documents
ID : 56c5fa40d36a4e7b72bfbef2
desc : greatest widget of all
title : First Great widget
price : 14.99
ID : 56c5fa40d36a4e7b72bfbef3
desc : second greatest widget of all
title : Second Great widget
price : 29.99
```

Você pode ver os registros no MongoDB usando a ferramenta de linha de comando. Utilize a sequência de comandos a seguir para iniciá-la e observar os registros:

1. Digite `mongo` para iniciar a ferramenta de linha de comando.
2. Digite `use exampleDb` para passar para o banco de dados `exampleDb`.
3. Digite `show collections` para ver todas as coleções.
4. Digite `db.widgets.find()` para ver os registros de Widget.

Se preferir uma abordagem mais orientada a objetos para incorporar o MongoDB à sua aplicação, utilize o Mongoose (*http://mongoosejs.com/*). Talvez ele seja mais adequado para integração com o Express.

Quando não estiver trabalhando com o MongoDB, lembre-se de encerrá-lo.

O MongoDB na documentação do Node

O driver do MongoDB para Node está documentado online e você poderá acessar a documentação a partir de seu repositório no GitHub (*https://github.com/mongodb/node-mongodb-native*). No entanto, você também poderá acessar a documentação do driver no site do MongoDB (*http://bit.ly/1W74pQC*). Prefiro a documentação do site do MongoDB, especialmente para as pessoas que estão começando a trabalhar com ele.

Redis: um repositório de chaves/valores

Quando se trata de dados, temos os bancos de dados relacionais e Tudo o Mais, também conhecidos como NoSQL. Na categoria NoSQL, um tipo de dado estruturado é baseado em pares chave/valor, geralmente armazenados em memória para um acesso extremamente rápido. Os três repositórios mais populares de pares chave/valor em memória são o Memcached, o Cassandra e o Redis. Felizmente para os desenvolvedores de Node, ele oferece suporte para todos os três repositórios.

O Memcached é usado principalmente como uma forma de caching para consulta de dados a fim de oferecer um acesso rápido em memória. Esse banco de dados também é muito apropriado para processamento distribuído, mas tem suporte limitado para dados mais complexos. É conveniente para aplicações que executam muitas consultas, mas nem tanto para aplicações com muita escrita e leitura de dados. O Redis é um repositório de dados melhor para o último tipo de aplicação. Além do mais, ele permite persistência e oferece mais flexibilidade que o Memcached – especialmente quanto ao suporte para diferentes tipos de dados. Contudo, de modo diferente do Memcached, o Redis funciona somente em um único computador.

Os mesmos fatores também entram em cena quando comparamos o Redis com o Cassandra. Assim como o Memcached, o Cassandra tem suporte para clusters. No entanto, também como o Memcached, ele oferece suporte limitado para estruturas de dados. O Cassandra é conveniente para consultas *ad hoc* – um uso que não favorece o Redis. No entanto, o Redis é fácil de usar, descomplicado e geralmente mais rápido que o Cassandra. Por esses e outros motivos, o Redis conquistou mais seguidores entre os desenvolvedores de Node.

Fiquei extremamente satisfeita ao ler o acrônimo EARN, isto é, Express, AngularJS, Redis e Node. Podemos ver um exemplo de EARN em "The EARN Stack" (A pilha EARN, *https://www.airpair.com/express/posts/earn-stack*).

Meu módulo Redis preferido do Node é instalado com o npm:

```
npm install redis
```

Se você planeja fazer grandes operações com o Redis, também recomendo instalar o módulo de suporte `hiredis` do Node, pois ele é não bloqueante e é capaz de melhorar o desempenho:

```
npm install hiredis redis
```

O módulo Redis é um wrapper relativamente leve em torno do Redis propriamente dito. Desse modo, você precisará investir tempo para conhecer os comandos do Redis e o modo como esse repositório de dados funciona.

Para usar o Redis em suas aplicações Node, inicialmente você deve incluir este módulo:

```
var redis = require('redis');
```

Em seguida, crie um cliente Redis. O método usado é o `createClient`:

```
var client = redis.createClient();
```

O método `createClient` aceita três parâmetros opcionais: `port`, `host` e opções (serão descritos em breve). Por padrão, o host é definido com `127.0.0.1` e a porta com `6379`. A porta é aquela usada por padrão em um servidor Redis, portanto essas configurações default deverão ser apropriadas se o servidor estiver hospedado na mesma máquina que a aplicação Node.

O terceiro parâmetro é um objeto que aceita várias opções e está descrito em detalhes na documentação do módulo. Utilize as configurações default até se sentir mais à vontade com o Node e o Redis.

Depois que tiver uma conexão entre o cliente e o repositório de dados Redis, você poderá enviar comandos ao servidor até chamar o método `client.quit()`, que encerrará a conexão com o servidor Redis. Se quiser forçar um encerramento, o método `client.end()` poderá ser usado no lugar do método anterior. No entanto, o último método não espera que o parse de todas as respostas seja feito. O método `client.end()` é uma boa opção a ser chamada caso a sua aplicação trave ou você queira recomeçar.

Executar comandos do Redis usando uma conexão de cliente é um processo relativamente intuitivo. Todos os comandos são expostos como métodos no objeto cliente e os argumentos dos comandos são passados como parâmetros. Como estamos usando Node, o último parâmetro é uma função de callback que devolve um erro e quaisquer dados ou resposta do comando para o Redis.

No código a seguir, o método `client.hset()` é usado para definir uma propriedade *hash*. No Redis, um hash é um mapeamento entre campos de string e valores, como, por exemplo, "lastname" para representar o seu sobrenome, "firstname" para o primeiro nome, e assim por diante:

```
client.hset("hashid", "propname", "propvalue", function(err, reply) {
  // faz algo com o erro ou com a resposta
});
```

O comando `hset` define um valor, portanto não há dado de retorno – somente a confirmação do Redis. Se você chamar um método que recupere diversos valores, como `client.hvals`, o segundo parâmetro da função de callback será um array – pode ser um array de strings simples ou um array de objetos:

```
client.hvals(obj.member, function (err, replies) {
  if (err) {
    return console.error("error response - " + err);
  }

  console.log(replies.length + " replies: ");
  replies.forEach(function (reply, i) {
    console.log(" " + i + ": " + reply);
  });
});
```

Como a callback do Node está presente em todos os lugares, e pelo fato de muitos dos comandos do Redis serem operações que simplesmente respondem com uma confirmação de sucesso, o módulo Redis oferece um método `redis.print` que pode ser passado como último parâmetro:

```
client.set("umachave", "umvalor", redis.print);
```

O método `redis.print` exibe o erro ou a resposta no console e retorna.

Para demonstrar o uso do Redis no Node, criarei uma *fila de mensagens*

(message queue). Uma fila de mensagens é uma aplicação que aceita alguma forma de comunicação como entrada, a qual é então armazenada em uma fila. As mensagens são armazenadas até que sejam recuperadas pelo receptor de mensagens, quando são removidas da fila e enviadas ao receptor (pode ser uma de cada vez ou em conjunto). A comunicação é assíncrona porque a aplicação que armazena as mensagens não exige que o receptor esteja conectado, e o receptor não exige que a aplicação esteja conectada.

O Redis é uma mídia de armazenagem ideal para esse tipo de aplicação. À medida que as mensagens são recebidas pela aplicação que as armazena, elas são inseridas no final da fila. Quando são recuperadas pela aplicação que as recebe, as mensagens são extraídas da frente da fila de mensagens.

Incorporando um pouco de TCP, HTTP e processos-filho

O exemplo com Redis incorpora também um servidor TCP (daí o motivo para trabalhar com o módulo Net do Node), um servidor HTTP, assim como um processo-filho. O Capítulo 5 descreve o HTTP, o Capítulo 7 discute o Net e o Capítulo 8 aborda os processos-filho.

Para uma demonstração da fila de mensagens, criei uma aplicação Node para acessar os arquivos de log web para vários subdomínios diferentes. A aplicação utiliza um processo-filho do Node e o comando `tail -f` do Unix para acessar as entradas recentes dos diferentes arquivos de log. Nessas entradas de log, a aplicação usa dois objetos que representam expressões regulares: um para extrair o recurso acessado e outro para testar se o recurso é um arquivo de imagem. Se o recurso acessado for um arquivo de imagem, a aplicação enviará o URL do recurso em uma mensagem TCP para a aplicação de fila de mensagens.

Tudo que a aplicação de fila de mensagens faz é ouvir mensagens de entrada na porta 3000 e armazenar o que quer que tenha sido enviado para um repositório de dados Redis.

A terceira parte da aplicação de demonstração é um servidor web que ouve requisições na porta 8124. A cada requisição, ela acessa o banco de dados Redis e extrai a entrada que está na frente no repositório de dados de imagens, devolvendo-a no objeto de resposta. Se o banco de dados

Redis devolver `null` para o recurso de imagem, uma mensagem informando que a aplicação alcançou o final da fila de mensagens será exibida.

Apresentamos a primeira parte da aplicação, que processa as entradas de log web, no Exemplo 10.2. O comando `tail` do Unix é uma forma de exibir as últimas linhas de um arquivo-texto (ou dos dados de um pipe). Quando usado com a flag `-f`, o utilitário exibe algumas linhas do arquivo e então aguarda novas entradas no arquivo. Quando uma nova entrada surgir, a nova linha será devolvida. O comando `tail -f` pode ser usado em vários arquivos diferentes ao mesmo tempo e administra o conteúdo inserindo um rótulo informando o local de onde os dados são provenientes sempre que houver uma origem distinta. A aplicação não está preocupada com o log de acesso que gerou a última resposta de `tail` – ela simplesmente quer a entrada do log.

Depois que a aplicação obtiver a entrada de log, ela executará duas correspondências com expressões regulares nos dados a fim de identificar um acesso a um recurso de imagem (arquivos com uma extensão *.jpg*, *.gif*, *.svg* ou *.png*). Se houver uma correspondência de padrão, a aplicação enviará o URL do recurso para a aplicação de fila de mensagens (um servidor TCP). A aplicação é simples, portanto não haverá uma verificação para garantir se a ocorrência da string corresponde realmente a uma extensão de arquivo e não a uma string incluída no nome desse arquivo, como *this.jpg.html*. Você poderá obter falso-positivos. No entanto, o exemplo faz uma demonstração do Redis, que é o que importa.

Exemplo 10.2 – Aplicação Node que processa entradas de log e envia requisições de recursos de imagens à fila de mensagens

```
var spawn = require('child_process').spawn;
var net = require('net');

var client = new net.Socket();
client.setEncoding('utf8');

// conecta-se com o servidor TCP
client.connect ('3000', 'examples.burningbird.net', function() {
  console.log('connected to server');
});
```

```
// inicia o processo-filho
var logs = spawn('tail', ['-f',
  '/home/main/logs/access.log',
  '/home/tech/logs/access.log',
  '/home/shelleypowers/logs/access.log',
  '/home/green/logs/access.log',
  '/home/puppies/logs/access.log']);

// processa os dados do processo-filho
logs.stdout.setEncoding('utf8');
logs.stdout.on('data', function(data) {

  // URL do recurso
  var re = /GET\s(\S+)\sHTTP/g;

  // testa se é imagem
  var re2 = /\.gif|\.png|\.jpg|\.svg/;

  // extrai o URL
  var parts = re.exec(data);
  console.log(parts[1]);

  // procura a imagem e, se ela for encontrada, armazena
  var tst = re2.test(parts[1]);
  if (tst) {
    client.write(parts[1]);
  }
});
logs.stderr.on('data', function(data) {
  console.log('stderr: ' + data);
});

logs.on('exit', function(code) {
  console.log('child process exited with code ' + code);
  client.end();
});
```

Podemos ver entradas típicas de log do console para essa aplicação no próximo bloco de código, com as entradas de interesse (os acessos aos arquivos de imagem) em negrito:

```
/robots.txt
/weblog
/writings/fiction?page=10
/images/kite.jpg
/node/145
/culture/book-reviews/silkworm
```

```
/feed/atom/
/images/visitmologo.jpg
/images/canvas.png
/sites/default/files/paws.png
/feeds/atom.xml
```

O Exemplo 10.3 mostra o código para a fila de mensagens. É uma aplicação simples, que inicia um servidor TCP e ouve mensagens de entrada. Quando uma mensagem é recebida, a aplicação extrai seus dados e os armazena no banco de dados Redis. A aplicação utiliza o comando `rpush` do Redis para inserir os dados no final da lista de imagens (em negrito no código).

Exemplo 10.3 – Fila de mensagens que aceita mensagens de entrada e as insere em uma lista do Redis

```
var net = require('net');
var redis = require('redis');

var server = net.createServer(function(conn) {
  console.log('connected');

  // cria um cliente Redis
  var client = redis.createClient();

  client.on('error', function(err) {
    console.log('Error ' + err);
  });

  // o sexto banco de dados é uma fila de imagens
  client.select(6);
  // ouve dados de entrada
  conn.on('data', function(data) {
    console.log(data + ' from ' + conn.remoteAddress + ' ' + conn.remotePort);

    // armazena dados
    client.rpush('images',data);
  });

}).listen(3000);
server.on('close', function(err) {
  client.quit();
});

console.log('listening on port 3000');
```

As entradas de log do console para a aplicação de fila de mensagens

geralmente terão o seguinte aspecto:

```
listening on port 3000
connected
/images/venus.png from 173.255.206.103 39519
/images/kite.jpg from 173.255.206.103 39519
/images/visitmologo.jpg from 173.255.206.103 39519
/images/canvas.png from 173.255.206.103 39519
/sites/default/files/paws.png from 173.255.206.103 39519
```

A última parte da aplicação para demonstração da fila de mensagens é o servidor HTTP que ouve requisições na porta 8124, como mostra o Exemplo 10.4. À medida que o servidor HTTP recebe cada requisição, ele acessa o banco de dados Redis, extrai a próxima entrada da lista de imagens e a escreve na resposta. Se não houver mais entradas na lista (isto é, o Redis devolveu `null` como resposta), será exibida uma mensagem informando que a fila de mensagens está vazia.

Exemplo 10.4 – Servidor HTTP que extrai mensagens da lista do Redis e devolve ao usuário

```
var redis = require("redis"),
  http = require('http');

var messageServer = http.createServer();

// ouve requisições de entrada
messageServer.on('request', function (req, res) {

  // inicialmente filtra a requisição de ícone
  if (req.url === '/favicon.ico') {
    res.writeHead(200, {'Content-Type': 'image/x-icon'} );
    res.end();
    return;
  }

  // cria um cliente Redis
  var client = redis.createClient();

  client.on('error', function (err) {
    console.log('Error ' + err);
  });

  // define o banco de dados para 6, que é a fila de imagens
  client.select(6);
  client.lpop('images', function(err, reply) {
```

```
    if(err) {
      return console.error('error response ' + err);
    }
    // se houver dados
    if (reply) {
      res.write(reply + '\n');
    } else {
      res.write('End of queue\n');
    }
    res.end();
  });
  client.quit();
});
messageServer.listen(8124);
console.log('listening on 8124');
```

Acessar a aplicação de servidor HTTP com um navegador web devolve um URL para o recurso de imagem a cada requisição (atualização do navegador), até que a fila de mensagens esteja vazia.

Os dados envolvidos são muito simples e, possivelmente, prolíficos, motivo pelo qual o Redis é ideal para esse tipo de aplicação. É um repositório de dados rápido e descomplicado, que não exige muito esforço para que o seu uso seja incorporado em uma aplicação Node.

Quando criar o cliente Redis

Em meu trabalho com o Redis, às vezes crio um cliente Redis e deixo-o persistente pelo tempo de vida da aplicação, enquanto em outras ocasiões crio um cliente Redis e o libero assim que o comando para o Redis termina. Quando é melhor criar uma conexão persistente com o Redis, em comparação com criar uma conexão e liberá-la de imediato?

Boa pergunta.

Para testar o impacto das duas abordagens, criei um servidor TCP que ouvia requisições e armazenava um hash simples no banco de dados Redis. Em seguida, criei outra aplicação como um cliente TCP que não fazia nada além de enviar um objeto em uma mensagem TCP para o servidor.

Usei a aplicação ApacheBench para executar várias iterações concorrentes do cliente e testei quanto tempo cada execução demorou. Executei o primeiro batch com uma conexão persistente do cliente Redis pelo tempo de vida do servidor e executei o segundo com uma conexão com o cliente criada a cada requisição e imediatamente liberada.

Esperava constatar que a aplicação com a conexão persistente com o cliente fosse mais

rápida, e eu estava certa... até certo ponto. Aproximadamente na metade do teste com a conexão persistente, a aplicação se tornou drasticamente mais lenta durante um breve período de tempo e, em seguida, retomou o seu ritmo relativamente rápido.

É claro que o mais provável que tenha acontecido é que as requisições enfileiradas para o banco de dados Redis em algum momento bloquearam a aplicação Node, pelo menos temporariamente, até que a fila estivesse livre. Não me deparei com essa mesma situação quando abri e fechei as conexões a cada requisição, pois o overhead extra exigido por esse processo deixou a aplicação lenta o suficiente para que o limite superior de usuários concorrentes não fosse atingido.

AngularJS e outros frameworks full-stack

Antes de tudo, o termo framework é genericamente usado de forma exagerada. Vemos esse termo ser usado para bibliotecas de frontend como jQuery, bibliotecas gráficas como D3, para o Express e uma gama de aplicações full-stack mais modernas. Neste capítulo, quando uso "framework", refiro-me a frameworks full-stack, como AngularJS, Ember e Backbone.

Para ter familiaridade com frameworks full-stack, você deverá conhecer o site TodoMVC (*http://todomvc.com/*). Esse site define os requisitos para um tipo básico de aplicação – uma lista de to-dos (tarefas a fazer) – e, então, convida todo e qualquer desenvolvedor de framework a submeter implementações para essa aplicação. O site também oferece uma implementação bem básica da aplicação, sem o uso de qualquer framework, assim como uma implementação com jQuery. Os desenvolvedores também têm uma forma de contrastar e comparar de que modo a mesma funcionalidade pode ser implementada em cada framework. Estão aí incluídos todos os frameworks populares, não apenas o AngularJS: Backbone.js, Dojo, Ember, React, e assim por diante. O site também inclui aplicações que incorporam várias tecnologias, como uma que utiliza AngularJS, Express e a Plataforma Google Cloud.

Os requisitos da aplicação To-Do incluem um diretório recomendado e uma estrutura de arquivos:

```
index.html
package.json
node_modules/
```

```
css
└── app.css
js/
├── app.js
├── controllers/
└── models/
readme.md
```

Não há nada inusitado nessa estrutura, e ela lembra o que vimos na aplicação ExpressJS. Porém, o modo como cada framework atende aos requisitos pode ser bem diferente, motivo pelo qual a aplicação ToDo é uma ótima forma de saber como cada framework funciona.

Para demonstrar, vamos dar uma olhada em parte do código usado com dois frameworks: AngularJS e Backbone.js. Não vou reproduzir boa parte do código, pois é quase certo que ele terá sofrido alterações até o momento em que você estiver lendo este livro. Começarei pelo AngularJS e me concentrarei na aplicação otimizada – o site inclui várias implementações diferentes com o AngularJS. A Figura 10.1 mostra a aplicação após três itens to-do terem sido adicionados.

Figura 10.1 – A aplicação To-Do após três itens de tarefas a fazer terem sido adicionados.

Para começar, a raiz da aplicação, *app.js*, é bem simples, conforme esperado, com todas as funcionalidades separadas nos vários subgrupos de modelo-visão-controlador.

```
/* jshint undef: true, unused: true */
/*global angular */
(function () {
'use strict';

/**
 * O módulo principal da aplicação TodoMVC que extrai todos os módulos
dependentes
 * declarados nos arquivos de mesmo nome
 *
 * @type {angular.Module}
 */
angular.module('todomvc', ['todoCtrl', 'todoFocus', 'todoStorage']);
})();
```

O nome da aplicação é `todomvc` e ela incorpora três serviços: `todoCtrl`, `todoFocus` e `todoStorage`. A interface de usuário está incluída no arquivo *index.html, localizado no diretório-raiz*. O arquivo é bem grande, portanto apresentarei apenas parte dele. O corpo da página principal está incluído em um elemento `section`, com a seguinte definição:

```
<section id="todoapp" ng-controller="TodoCtrl as TC">
...
</section>
```

O AngularJS adiciona anotações ao HTML chamadas de *diretivas* (derivatives). Você as reconhecerá facilmente, pois as diretivas-padrão começam com “ng-”, como `ng-submit`, `ng-blur` e `ng-model`. No trecho de código anterior, a diretiva `ng-controller` define o controlador para a visão (view), isto é, `TodoCtrl`, e a referência usada para ele, que aparecerá em outros lugares no template, isto é, TC.

```
<form ng-submit="TC.doneEditing(todo, $index) ">
  <input class="edit"
    ng-trim="false"
    ng-model="todo.title"
    ng-blur="TC.doneEditing(todo, $index)"
    ng-keydown="($event.keyCode === TC.ESCAPE_KEY)
       && TC.revertEditing($index)"
```

```
    todo-focus="todo === TC.editedTodo">
</form>
```

Podemos ver várias diretivas em funcionamento, e o seu propósito é intuitivo na maioria das vezes. É na diretiva `ng-model` que a visão encontra o modelo (dados), nesse caso, `todo.title`. As funções `TC.doneEditing` e `TC.revertEditing` são os controladores. Extraí seus códigos do arquivo de controlador e os reproduzi a seguir. A função `TC.doneEditing` reinicia o objeto `TC.editedTodo`, remove espaços excedentes do título do to-do editado e, se não houver nenhum título, remove o to-do. A função `TC.revertEditing` também reinicia o objeto to-do e atribui o to-do original novamente ao seu índice no array de to-dos.

```
TC.doneEditing = function (todo, index) {
  TC.editedTodo = {};
  todo.title = todo.title.trim();

  if (!todo.title) {
    TC.removeTodo(index);
  }
};

TC.revertEditing = function (index) {
  TC.editedTodo = {};
  todos[index] = TC.originalTodo;
};
```

Não há nada excessivamente complexo nesse código. Faça um ckeckout de uma cópia do código (localizado no GitHub em *https://github.com/tastejs/todomvc/tree/gh-pages/examples/angularjs-perf*) e teste-o por conta própria.

A aplicação com Backbone.js se parece e se comporta da mesma forma que a aplicação com o AngularJS, mas o código-fonte é bem diferente. Embora o arquivo *app.js* do AngularJS não seja muito grande, o arquivo do *Backbone.js* é menor ainda:

```
/*global $ */
/*jshint unused:false */
var app = app || {};
var ENTER_KEY = 13;
var ESC_KEY = 27;

$(function () {
```

```
'use strict';

// dá a partida criando a `App`
new app.AppView();
});
```

Basicamente, `app.AppView()` inicia a aplicação. O arquivo *app.js* é simples, mas a implementação de `app.AppView()` não é. Em vez de fazer anotações no HTML com diretivas, como faz o AngularJS, o Backbone.js faz uso intenso de templates Underscore. No arquivo *index.html*, você verá o seu uso em elementos de script in-page, como no código a seguir, que representa o template para cada to-do individual. Intercaladas no HTML estão as tags de template, como `title`, e se a caixa de seleção está marcada ou não.

```
<script type="text/template" id="item-template">
  <div class="view">
    <input class="toggle" type="checkbox" <%= completed ? 'checked'
      : '' %>>
    <label><%- title %></label>
    <button class="destroy"></button>
  </div>
  <input class="edit" value="<%- title %>">
</script>
```

A renderização dos itens ocorre no arquivo *todo-view.js*, mas a força motriz por trás da renderização está no arquivo *app-view.js*. Apresentarei a seguir uma parte desse arquivo:

```
// Adiciona um único item to-do à lista criando uma visão para ela e
// concatenando o seu elemento no `<ul>`.
addOne: function (todo) {
  var view = new app.TodoView({ model: todo });
  this.$list.append(view.render().el);
},
```

A renderização ocorre no arquivo *todo-view.js*; mostraremos parte dele a seguir. Você pode ver a referência ao identificador de item da lista `item-template`, especificado anteriormente no script incluído no arquivo *index.html*. O HTML no elemento script do arquivo *index.html* fornece o template para os itens renderizados pela visão. No template, há um placeholder para os dados fornecidos pelo modelo na aplicação. Na

aplicação To-Do, os dados são o título do item e se a tarefa foi concluída ou não.

```
// O elemento do DOM para um item de to-do...
app.TodoView = Backbone.View.extend({
  //... é uma tag de lista.
  tagName: 'li',

  // Faz cache da função de template para um único item.
  template: _.template($('#item-template').html()),

  ...

  // Renderiza novamente os títulos do item to-do.
  render: function () {
    // O LocalStorage do Backbone adiciona o atributo `id` imediatamente após
    // criar um modelo. Isso faz com que o nosso TodoView renderize duas vezes.
    // Uma vez após criar um modelo e outra na mudança de `id`. Queremos
    // filtrar a segunda renderização redundante, provocada por essa
    // mudança de `id`. É um bug conhecido do LocalStorage do Backbone,
portanto
    // devemos criar uma solução para contorná-lo.
    // https://github.com/tastejs/todomvc/issues/469
    if (this.model.changed.id !== undefined) {
      return;
    }

    this.$el.html(this.template(this.model.toJSON()));
    this.$el.toggleClass('completed', this.model.get('completed'));
    this.toggleVisible();
    this.$input = this.$( '.edit');
    return this;
  },
```

É um pouco difícil acompanhar o que está acontecendo no Backbone.js em comparação com o AngularJS, mas, assim como no caso anterior, trabalhar com a aplicação To-Do possibilita uma boa dose de clareza. Sugiro também que você dê uma olhada na variante que está em *https://github.com/tastejs/todomvc/tree/gh-pages/examples/backbone* e experimente usá-la.

A renderização da visão é apenas uma das diferenças entre os frameworks. O AngularJS reconstrói o DOM quando uma mudança ocorre, enquanto o Backbone.js faz as alterações in-place. O AngularJS oferece vinculação

de dados bidirecional (two-way data binding), o que significa que as mudanças na UI e no modelo são automaticamente sincronizadas. A arquitetura do Backbone.js é MVP (Model-View-Presenter, ou Modelo-Visão-Apresentador), enquanto a do AngularJS é MVC (Model-View-Controller, ou Modelo-Visão-Controlador), o que implica que o Backbone.js não oferece o mesmo tipo de vinculação de dados e você deverá desenvolvê-lo por conta própria. Por outro lado, o Backbone.js é mais leve e pode ser mais rápido que o AngularJS, embora este geralmente seja mais simples de ser compreendido por desenvolvedores que estejam começando a trabalhar com frameworks.

Esses dois e os demais frameworks full-stack são usados para criar páginas web dinamicamente. Com isso, não estou me referindo ao mesmo tipo de geração dinâmica de página explorada anteriormente na seção "Framework de aplicação Express". Esses frameworks possibilitam criar um tipo específico de aplicação conhecida como *aplicação single-page* (de página única), ou *SPA*. Em vez de gerar o HTML no servidor e enviá-lo para o navegador, os dados são empacotados, enviados ao navegador e então a página web é formatada usando JavaScript.

A vantagem desse tipo de funcionalidade está no fato de a página web nem sempre precisar ser atualizada quando você muda a visão dos dados ou explora mais detalhes na página.

Considere a aplicação Gmail. Ela é um exemplo de uma SPA. Quando você abre a caixa de entrada da aplicação e acessa um dos emails, a página toda não é recarregada. Em vez disso, qualquer dado necessário ao email será recuperado do servidor e incorporado na exibição da página. O resultado é mais rápido e causa menos perturbações ao usuário. Contudo, não queira jamais olhar o código-fonte da página. Se quiser preservar a sua sanidade, *nunca* use o recurso de código-fonte da página em seu navegador para ver as páginas do Google.

O que um bom framework deve oferecer? Em minha opinião, um dos recursos para os quais os frameworks devem oferecer suporte é a vinculação de dados entre a exibição e os dados. Isso significa que se os dados mudarem, a interface do usuário deverá ser atualizada. O

framework também deve ter suporte para uma engine de templates, como o Jade usado antes em conjunto com o Express. Também deve oferecer uma maneira de reduzir códigos redundantes, portanto o framework deve prover suporte para componentes reutilizáveis e/ou modularização.

Na aplicação com Express, vimos uma associação entre roteamento de URL e funções. O URL passa a ser a única forma de acessar grupos de dados ou itens únicos. Para encontrar um único aluno, você poderia ter um URL como */students/A1234*, e a requisição seria encaminhada para a página com detalhes sobre o aluno identificado por `A1234`. Os frameworks devem oferecer suporte para esse tipo de roteamento.

O framework também deve oferecer suporte para um esquema MV*, o que significa que, no mínimo, a lógica de negócios deve estar separada da lógica de visualização. Ele poderia ter suporte para uma variação do modelo MVC (modelo-visão-controlador), MVP (modelo-visão-apresentador), MVVM (modelo-visão-visão-modelo), e assim por diante, mas, no mínimo, deve haver uma separação entre os dados e a UI.

E, é claro, considerando o contexto deste livro, deverá ser possível integrar o framework ao Node.

CAPÍTULO 11

Node nos ambientes de desenvolvimento e de produção

O surgimento do Node coincidiu com (e até mesmo inspirou) uma série de novas ferramentas e técnicas para desenvolvimento, gerenciamento e manutenção de código. Depuração, testes, gerenciamento de tarefas, rollout em produção e suporte são elementos essenciais em qualquer projeto Node, e é bom que a maioria deles esteja automatizada.

Este capítulo apresenta algumas das ferramentas e conceitos associados a essas tarefas. Não é uma lista exaustiva, mas deve oferecer um bom ponto de partida para suas explorações.

Depurando aplicações Node

Confesso que utilizo logging no console mais do que deveria para depuração. É uma abordagem fácil para conferir valores de variáveis e resultados. No entanto, o problema de usar o console é que afetamos a dinâmica e o comportamento da aplicação, e poderíamos, na verdade, mascarar – ou criar – um problema simplesmente por usá-lo. É realmente melhor utilizar ferramentas de depuração, especialmente quando a aplicação crescer e tiver mais do que blocos de código simples.

O Node oferece um depurador embutido que podemos usar para definir breakpoints (pontos de parada) no código e adicionar watchers (observadores) a fim de visualizar resultados de códigos intermediários. Não é a ferramenta mais sofisticada do mundo, mas é suficiente para identificar bugs e possíveis complicações. Na próxima seção, veremos uma ferramenta mais sofisticada: o Node Inspector.

Depurador do Node

Se for possível escolher, sempre prefiro implementações nativas no lugar de funcionalidades de terceiros. Felizmente, para as nossas necessidades de depuração, o Node oferece suporte na forma de um depurador embutido. Ele não é sofisticado, mas pode ser útil.

É possível definir breakpoints inserindo o comando `debugger` diretamente em seu código:

```
for (var i = 0; i <= test; i++) {
  debugger;
  second+=i;
}
```

Para iniciar a depuração da aplicação, especifique a opção `debug` ao executá-la:

```
node debug application
```

Com o intuito de fazer uma demonstração do depurador, criei uma aplicação com dois breakpoints `debugger` incluídos:

```
var fs = require('fs');
var concat = require('./external.js').concatArray;

var test = 10;
var second = 'test';

for (var i = 0; i <= test; i++) {
  debugger;
  second+=i;
}

setTimeout(function() {
  debugger;
  test = 1000;
  console.log(second);
}, 1000);

fs.readFile('./log.txt', 'utf8', function (err,data) {
  if (err) {
    return console.log(err);
  }
  var arry = ['apple', 'orange', 'strawberry'];
  var arry2 = concat(data,arry);
  console.log(arry2);
```

```
});
```

A aplicação é iniciada com o comando a seguir:

```
node debug debugtest
```

Se você iniciar uma aplicação Node com a flag de linha de comando `--debug`, poderá também iniciar o depurador fazendo a conexão com um processo usando o pid:

```
node debug -p 3383
```

Ou poderá iniciá-lo fazendo uma conexão com o processo em execução por meio do URI:

```
node debug http://localhost:3000
```

Quando o depurador for disparado, a aplicação fará uma pausa na linha 1 e listará a parte inicial do código:

```
< Debugger listening on port 5858
debug> . ok
break in debugtest.js:1
> 1 var fs = require('fs');
  2 var concat = require('./external.js').concatArray;
  3
```

Você pode usar o comando `list` para listar as linhas do código-fonte no contexto. Um comando como `list(10)` listará as dez linhas anteriores e as próximas dez linhas de código. Digitar `list(25)` com a aplicação de teste de depuração fará todas as linhas da aplicação serem exibidas, numeradas por linha. Você pode adicionar outros breakpoints nesse ponto usando o comando `setBreakpoint` ou a sua forma abreviada `sb`. Definiremos um breakpoint na linha 19 da aplicação de teste, que inserirá um breakpoint na função de callback `fs.readFile()`. Também definiremos um breakpoint diretamente no módulo personalizado, na linha 3:

```
debug> sb(19)
debug> sb('external.js',3)
```

Você verá um aviso sobre o script *external.js* ainda não estar carregado. Isso não terá impacto na funcionalidade.

Também é possível definir um watch em variáveis ou expressões usando o comando `watch('expression') diretamente no depurador.` Observaremos as

variáveis `test` e `second`, o parâmetro `data` e o array `arry2`:

```
debug> watch('test');
debug> watch('second');
debug> watch('data');
debug> watch('arry2');
```

Finalmente estamos prontos para dar início à depuração. Digitar `cont` ou `c` executará a aplicação até que o primeiro breakpoint seja alcançado. Na saída, podemos notar que estamos no primeiro breakpoint e ver também o valor dos quatro itens observados. Dois deles – `test` e `second` – têm realmente um valor, enquanto os outros dois têm valor igual a `<error>`. Isso ocorre porque a aplicação no momento está fora do escopo das funções em que o parâmetro (`data`) e a variável (`arry2`) estão definidos. Siga em frente e ignore os erros por enquanto.

```
debug> c
break in debugtest.js:8
Watchers:
  0: test = 10
  1: second = "test"
  2: data = "<error>"
  3: arry2 = "<error>"

  6
  7 for (var i = 0; i <= test; i++) {
> 8 debugger;
  9 second+=i;
 10 }
```

Há alguns comandos que você pode experimentar antes de prosseguir para o próximo breakpoint. O comando `scripts` lista os scripts que estão `carregados no momento:`

```
debug> scripts
* 57: debugtest.js
  58: external.js
debug>
```

O comando `version` exibe a versão do V8. Digite `c` novamente para avançar até o próximo breakpoint:

```
debug> c
break in debugtest.js:8
Watchers:
```

```
  0: test = 10
  1: second = "test0"
  2: data = "<error>"
  3: arry2 = "<error>"

  6
  7 for (var i = 0; i <= test; i++) {
> 8 debugger;
  9 second+=i;
 10 }
```

Observe o valor alterado da variável `second`. Isso se deve ao fato de ela ser modificada no laço `for` em que o depurador está. Digitar `c` mais algumas vezes executará o laço e poderemos ver a variável ser modificada continuamente. Infelizmente, não podemos limpar o breakpoint criado com a instrução `debugger`, mas podemos remover um breakpoint definido com `setBreakpoint` ou `sb`. Para usar `clearBreakpoint` ou `cb`, especifique o nome do script e o número da linha que contém o breakpoint:

```
cb('debugtest.js',19)
```

Você também pode desativar um watcher usando `unwatch`:

```
debug> unwatch('second')
```

Utilizar `sb` sem nenhum valor define um breakpoint na linha atual:

```
debug> sb();
```

Na aplicação, o depurador executará até o próximo breakpoint, que está na callback `fs.readFile()`. Agora veremos que o parâmetro `data` mudou:

```
debug> c
break in debugtest.js:19
Watchers:
  0: test = 10
  1: second = "test012345678910"
  2: data = "test"
  3: arry2 = undefined

 17
 18 fs.readFile('./log.txt', 'utf8', function (err,data) {
>19 if (err) {
 20 return console.log(err);

 21 }
```

Podemos ver também que o valor de `arry2` não é mais um erro, mas

`undefined`.

Em vez de digitar `c` para continuar, executaremos cada linha da aplicação usando o comando `next` ou `n`. Quando chegarmos à linha 23, o depurador abrirá o módulo externo e fará uma pausa na linha 3 por causa do breakpoint definido anteriormente:

```
debug> n
break in external.js:3
Watchers:
  0: test = "<error>"
  1: second = "<error>"
  2: data = "<error>"
  3: arry2 = "<error>"

  1
  2 var concatArray = function(str, arry) {
> 3 return arry.map(function(element) {
  4 return str + ' ' + element;
  5 });
```

Poderíamos ter também executado até a linha 23 da aplicação e usado o comando `step` ou `s` para *entrar* na função do módulo:

```
debug> s
break in external.js:3
Watchers:
  0: test = "<error>"
  1: second = "<error>"
  2: arry2 = "<error>"
  3: data = "<error>"

  1
  2 var concatArray = function(str, arry) {
> 3 return arry.map(function(element) {
  4 return str + ' ' + element;
  5 });
```

Veja agora como todos os valores observados exibem um erro. Nesse ponto, estamos totalmente fora do contexto da aplicação-pai. Podemos adicionar watchers quando estivermos em funções ou em módulos externos para evitar esses tipos de erros ou para observar as variáveis no contexto em que elas são definidas se a mesma variável for usada na aplicação e no módulo ou em outras funções.

Bug do comando continue

Se você digitar `c` ou `cont` e a aplicação estiver no final, o depurador travará. Nada fará você sair daí. Esse é um bug conhecido.

O comando `backtrace` ou `bt` oferece um *backtrace* `do contexto da execução no momento`. O próximo bloco de código mostra o valor devolvido nesse ponto da depuração:

```
debug> bt
#0 concatArray external.js:3:3
#1 debugtest.js:23:15
```

Podemos ver duas entradas: uma para a linha em que estamos agora na aplicação e outra que representa a linha atual na função do módulo importado.

Podemos executar os passos da função externa ou retornar à aplicação usando o comando `out` ou `o`. Esse comando o faz retornar à aplicação quando estiver em uma função (independentemente de essa função estar em um script local ou em um módulo).

O depurador do Node é baseado em REPL, e podemos obter o REPL do depurador digitando o comando `repl`. Se quiser matar o script, o comando `kill` poderá ser usado; se quiser reiniciá-lo, digite `restart`. Todavia, esteja ciente de que o script não só será reiniciado, mas todos os breakpoints e watchers serão limpos.

Como o depurador executa em REPL, Ctrl-C encerrará a aplicação, ou você poderá digitar `.exit`.

Node Inspector

Se estiver pronto para dar um passo além em sua depuração, você vai querer dar uma olhada no Node Inspector. Ele incorpora recursos de depuração com os quais você provavelmente tem familiaridade usando o depurador Blink DevTools que funciona tanto com o Chrome quanto com o Opera. A vantagem da ferramenta é que ela apresenta um nível elevado de sofisticação e tem um ambiente visual para depuração. Sua desvantagem são os requisitos de sistema. Por exemplo, para executá-la em um de meus computadores com Windows 10, a aplicação informou

que eu deveria instalar o .NET Framework 2.0 SDK ou o Microsoft Visual Studio 2005.

Instalando o Node Inspector

Se você se deparar com o erro sobre o Visual Studio, poderá tentar fazer o seguinte para alterar a versão do Visual Studio usado pela aplicação:

```
npm install -g node-inspector --msvs_version=2013
```

Ou utilize qualquer versão que tenha instalado.

Se você tiver um ambiente capaz de oferecer suporte ao Node Inspector, instale-o com:

```
npm install -g node-inspector
```

Para usá-lo, execute a sua aplicação Node com o comando a seguir. Será necessário acrescentar a extensão *.js* à aplicação quando executá-la com o Node Inspector:

```
node-debug application.js
```

Dois cenários serão possíveis. Se o Chrome ou o Opera for o seu navegador-padrão, a aplicação abrirá nas ferramentas de desenvolvedor. Se o seu navegador-padrão não for nenhum deles, será necessário abrir manualmente um dos navegadores e fornecer o URL *http://127.0.0.1:8080/?port=5858*.

Documentação do Node Inspector

Há alguma documentação sobre o Node Inspector em seu repositório no GitHub (*https://github.com/node-inspector/node-inspector*). A Strongloop – empresa que dá suporte à ferramenta – também fornece alguma documentação (*http://bit.ly/1OQEMkB*) sobre o uso do Node Inspector. Conforme consta na documentação, você também poderá obter as informações de que precisar na documentação do Chrome Developer Tools (Ferramentas de desenvolvedor do Chrome; *http://bit.ly/25e082Z*). Porém, esteja ciente de que o Google frequentemente altera o local em que está a documentação e ela poderá não estar funcionando ou estar incompleta a qualquer momento.

Abri o meu arquivo *debugtest.js* criado na última seção no Node Inspector. A Figura 11.1 mostra o depurador com o arquivo inicialmente carregado e após clicar o botão de execução, localizado no canto superior direito da ferramenta. O Node Inspector trata o comando `debugger`, que passa a ser o primeiro breakpoint da aplicação. As variáveis observadas são exibidas logo abaixo dos botões para controlar a execução do programa.
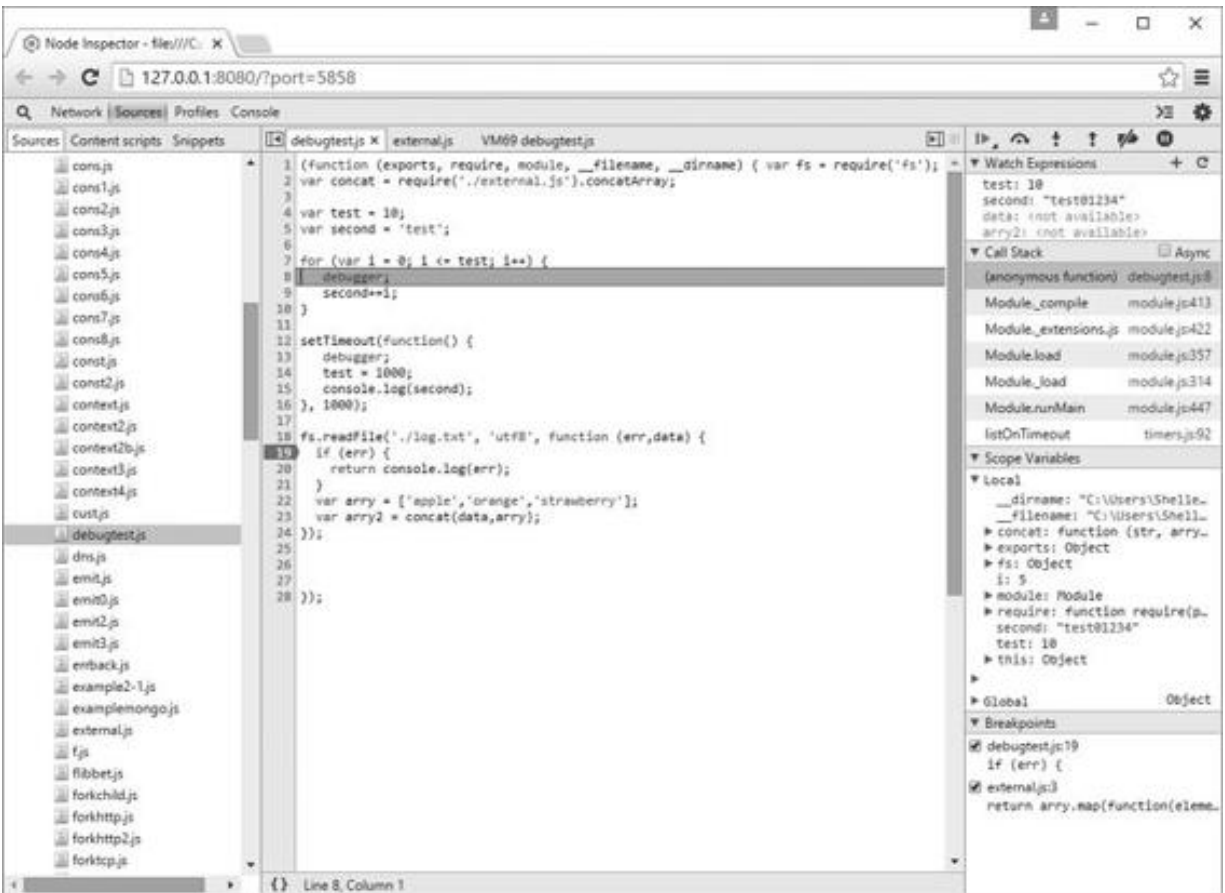
Figura 11.1 – Executando debugtest.js no depurador Node Inspector.

Definir novos breakpoints é simplesmente uma questão de clicar na margem esquerda da linha na qual você deseja definir o breakpoint. Para adicionar um novo watcher, clique no sinal de mais (+) no cabeçalho Watch Expressions (Observar expressões). Os comandos do programa na parte superior permitem (na sequência): executar a aplicação até o próximo breakpoint, passar pela próxima chamada de função (step over), entrar na função (step into), sair da função atual (step out), limpar todos os breakpoints e fazer uma pausa na aplicação.

As novidades nessa maravilhosa interface visual são: a lista das aplicações/módulos na janela à esquerda, uma pilha de chamadas (call stack), uma lista das variáveis do escopo (tanto local quanto global) e os breakpoints definidos, nas janelas à direita. Se quiser adicionar um breakpoint no módulo importado *external.js*, basta abrir o arquivo a partir da lista à esquerda e inserir um breakpoint. A Figura 11.2 mostra o depurador com o módulo carregado, após alcançar o breakpoint.

O interessante ao observar a aplicação carregada no Node Inspector é que podemos ver como a aplicação está encapsulada em uma função anônima, conforme descrito no Capítulo 3, na seção "Como o Node encontra um módulo e o carrega na memória".
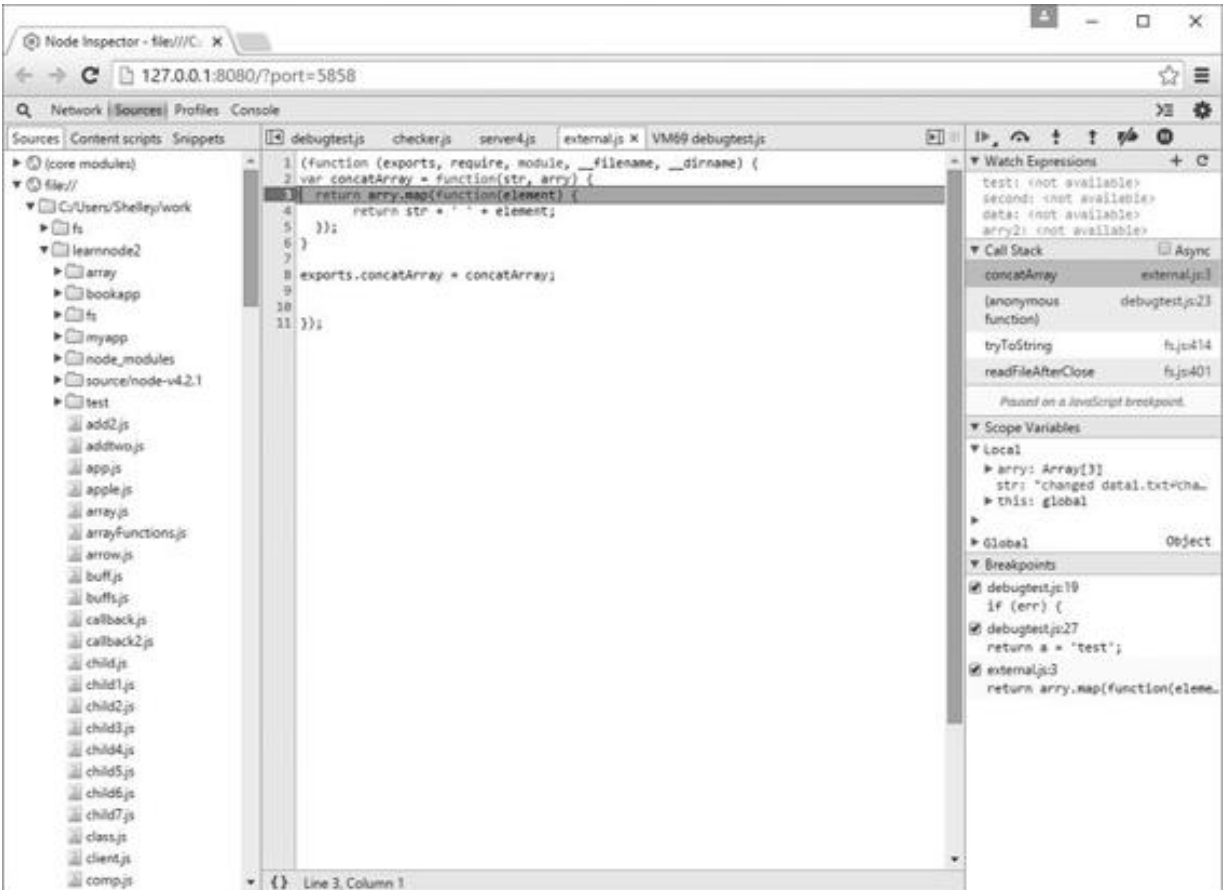
Figura 11.2 – Node Inspector com o módulo external.js e após alcançar o breakpoint.

Ao terminar de usar o Node Inspector, você poderá fechar o navegador e encerrar a operação em lote teclando Ctrl-C.

Ferramentas mais sofisticadas

Tanto o depurador embutido quanto o Node Inspector oferecem funcionalidades sólidas para ajudar a depurar as suas aplicações Node. Além disso, se quiser investir tempo na configuração do ambiente e adquirir familiaridade com um IDE (integrated development environment, ou ambiente de desenvolvimento integrado) mais sofisticado, você também poderá usar uma ferramenta como o Nodeclipse (*http://www.nodeclipse.org/usage*) no venerável Eclipse IDE.

Testes de unidade

Testes de unidade (unit testing) são uma maneira de isolar componentes específicos de uma aplicação com vistas a testes. Muitos dos testes disponibilizados no subdiretório *tests* dos módulos do Node são testes de unidade. Os testes no subdiretório *test* da instalação do Node são *todos* testes de unidade. Muitos deles são criados usando o módulo Assert, que veremos em seguida.

Testes de unidade com o Assert

Os testes de asserção (assertion tests) avaliam expressões cujo resultado final é um valor que pode ser `true` ou `false`. Se você estiver testando o valor de retorno de uma chamada de função, poderá inicialmente testar se é um array (primeira asserção). Se o conteúdo do array precisar ter determinado tamanho, você poderá executar um teste condicional nesse tamanho (segunda asserção), e assim por diante. Há um módulo embutido no Node que facilita essa forma de teste de asserção: o Assert. Ele foi criado para uso interno com o Node, mas podemos utilizá-lo. Basta estarmos cientes de que ele não é um verdadeiro framework para testes.

Inclua o módulo Assert em uma aplicação da seguinte maneira:

```
var assert = require('assert');
```

Para ver como usar o Assert, vamos observar como os módulos existentes o utilizam. A aplicação Node faz uso do módulo Assert nos testes de unidade de seus módulos. Por exemplo, há uma aplicação de teste chamada *test-util.js* que testa o módulo Utilities. O código a seguir é a seção que testa o método `isArray`:

```
// isArray
assert.equal(true, util.isArray([]));
assert.equal(true, util.isArray(Array()));
assert.equal(true, util.isArray(new Array()));
assert.equal(true, util.isArray(new Array(5)));
assert.equal(true, util.isArray(new Array('with', 'some', 'entries')));
assert.equal(true, util.isArray(context('Array')()));
assert.equal(false, util.isArray({}));
```

```
assert.equal(false, util.isArray({ push: function() {} }));
assert.equal(false, util.isArray(/regexp/));
assert.equal(false, util.isArray(new Error));
assert.equal(false, util.isArray(Object.create(Array.prototype)));
```

Tanto o método `assert.equal()` quanto o `assert.strictEqual()` têm dois parâmetros obrigatórios: uma resposta esperada e uma expressão que é avaliada como resposta. Em `assert.equal` de `isArray`, se a expressão for avaliada como `true` e a resposta esperada for `true`, o método terá sucesso e não produzirá nenhuma saída – o resultado será *silêncio*.

Porém, se a expressão for avaliada com uma resposta diferente daquela esperada, o método `assert.equal` responderá com uma exceção. Se tomarmos a primeira instrução do teste de `isArray` no código-fonte do Node e modificarmos para:

```
assert.equal(false, util.isArray([]));
```

o resultado será este:

```
assert.js:89
  throw new assert.AssertionError({
  ^
AssertionError: false == true
    at Object.<anonymous> (/home/examples/public_html/
                         learnnode2/asserttest.js:4:8)
    at Module._compile (module.js:409:26)
    at Object.Module._extensions..js (module.js:416:10)
    at Module.load (module.js:343:32)
    at Function.Module._load (module.js:300:12)
    at Function.Module.runMain (module.js:441:10)
    at startup (node.js:134:18)
    at node.js:962:3
```

Os métodos `assert.equal()` e `assert.strictEqual()` também têm um terceiro parâmetro opcional: uma mensagem que será exibida no lugar da mensagem default em caso de falha:

```
assert.equal(false, util.isArray([]), 'Test 1Ab failed');
```

Essa pode ser uma maneira útil de identificar exatamente qual teste falhou se você estiver executando vários em um script de teste. Podemos ver o uso de uma mensagem (um rótulo) no código de teste do node-redis:

```
assert.equal(str, results, label + " " + str +
             " does not match " + results);
```

A mensagem é exibida quando capturamos a exceção e exibimos a mensagem.

Todos os métodos seguintes do módulo Assert aceitam os mesmos três parâmetros, embora o modo como o valor do teste e a expressão se relacionam um com o outro varie, conforme indica o nome do teste:

`assert.equal`
: Falha se o resultado da expressão e o valor dado não forem iguais.

`assert.strictEqual`
: Falha se o resultado da expressão e o valor dado não forem estritamente iguais.

`assert.notEqual`
: Falha se o resultado da expressão e o valor dado forem iguais.

`assert.notStrictEqual`
: Falha se o resultado da expressão e o valor dado forem estritamente iguais.

`assert.deepEqual`
: Falha se o resultado da expressão e o valor dado não forem iguais.

`assert.notDeepEqual`
: Falha se o resultado da expressão e o valor dado forem iguais.

`assert.deepStrictEqual`
: Semelhante a `assert.deepEqual()`, exceto que as primitivas são comparadas como estritamente iguais (`===`).

`assert.notDeepStrictEqual`
: Testa se não são estritamente iguais de forma profunda.

Os métodos profundos (deep) trabalham com objetos complexos, como arrays ou objetos. O teste a seguir resulta em sucesso com `assert.deepEqual`:

```
assert.deepEqual([1,2,3],[1,2,3]);
```

mas não obterá sucesso com `assert.equal`.

Os métodos `assert` restantes aceitam parâmetros diferentes. Chamar `assert` como um método e passar um valor e uma mensagem é equivalente a chamar `assert.isEqual` e passar `true` como primeiro parâmetro, uma expressão e uma mensagem. As linhas a seguir:

```
var val = 3;
assert(val == 3, 'Test 1 Not Equal');
```

são equivalentes a:

```
assert.equal(true, val == 3, 'Test 1 Not Equal');
```

Ou utilize o alias (apelido) `assert.ok`:

```
assert.ok(val == 3, 'Test 1 Not Equal');
```

O método `assert.fail` lança uma exceção. Ele aceita quatro parâmetros: um valor, uma expressão, uma mensagem e um operador, usado para separar o valor e a expressão na mensagem quando uma exceção é lançada. No trecho de código a seguir:

```
try {
  var val = 3;
  assert.fail(val, 4, 'Fails Not Equal', '==');
} catch(e) {
  console.log(e);
}
```

a mensagem no console será:

```
{ [AssertionError: Fails Not Equal]
  name: 'AssertionError',
  actual: 3,
  expected: 4,
  operator: '==',
  message: 'Fails Not Equal',
  generatedMessage: false }
```

A função `assert.ifError` aceita um valor e lança uma exceção somente se o valor não for resolvido como `false`. Como descrito na documentação do Node, é um bom teste para o objeto de erro como o primeiro argumento em uma função de callback:

```
assert.ifError(err); //lança exceção somente se o valor for true
```

Os últimos métodos `assert` são `assert.throws` e `assert.doesNotThrow`. O primeiro espera que uma exceção seja lançada; o segundo não espera. Os dois métodos aceitam um bloco de código como primeiro parâmetro obrigatório; um erro opcional e uma mensagem correspondem ao segundo e ao terceiro parâmetros. O objeto erro pode ser um construtor, uma expressão regular ou uma função de validação. No trecho de código a seguir, a mensagem de erro é exibida porque a expressão regular de erro como segundo parâmetro não corresponde à mensagem de erro:

```
assert.throws(
  function() {
    throw new Error("Wrong value");
  },
  /algo/
);
```

Você pode criar testes de unidade robustos com o módulo Assert. Porém, uma limitação importante do módulo está no fato de você precisar fazer muitos encapsulamentos nos testes para que todo o script de testes não falhe caso um teste falhe. É em casos assim que usar um framework de testes de unidade de mais alto nível como o Nodeunit (discutido a seguir) se torna prático.

Testes de unidade com o Nodeunit

O Nodeunit oferece uma forma de criar scripts para vários testes. Depois de escritos, cada teste é executado serialmente e os resultados são informados de forma coordenada. Para usar o Nodeunit, você deve instalá-lo globalmente usando o npm:

```
[sudo] npm install nodeunit -g
```

O Nodeunit oferece uma maneira de executar facilmente uma série de testes sem ter de encapsular tudo em blocos try/catch. Ele aceita todos os testes do módulo Assert, além de oferecer um par de métodos próprios para controlar os testes. Os testes são organizados na forma de casos de teste, em que cada um é exportado como um método de objeto no script de teste. Cada caso de teste recebe um objeto de controle, geralmente chamado de `test`. A primeira chamada de método no caso de teste é para

o método `expect` do elemento `test` para dizer ao Nodeunit quantos testes devem ser esperados nesse caso de teste. A última chamada de método no caso de teste é para o método `done` do elemento `test` para dizer ao Nodeunit que o caso de teste terminou. Tudo que estiver entre eles compõe o teste de unidade propriamente dito:

```
module.exports = {
  'Test 1' : function(test) {
    test.expect(3); // três testes
    ... // os testes
    test.done();
  },
  'Test 2' : function (test) {
    test.expect(1); // apenas um teste
    ... // o teste
    test.done();
  }
};
```

Para executar os testes, digite `nodeunit`, seguido do nome do script de teste:

```
nodeunit thetest.js
```

O Exemplo 11.1 mostra um script de testes pequeno, porém completo, com seis asserções. Ele é constituído de duas unidades de testes, chamadas de `Test 1` e `Test 2`. A primeira unidade de teste executa quatro testes separados, enquanto a segunda executa dois. A chamada do método `expect` reflete a quantidade de testes executada na unidade.

Exemplo 11.1 – Script de teste do Nodeunit com duas unidades de testes, executando um total de seis testes

```
var util = require('util');

module.exports = {
  'Test 1' : function(test) {
    test.expect(4);
    test.equal(true, util.isArray([]));
    test.equal(true, util.isArray(new Array(3)));
    test.equal(true, util.isArray([1,2,3]));
    test.notEqual(true, 1 > 2);
    test.done();
  },
  'Test 2' : function(test) {
```

```
    test.expect(2);
    test.deepEqual([1,2,3], [1,2,3]);
    test.ok('str' === 'str', 'equal');
    test.done();
  }
};
```

Eis o resultado da execução do script de teste do Exemplo 11.1 com o Nodeunit:

```
thetest.js
✓ Test 1
✓ Test 2

OK: 6 assertions (12ms)
```

Os símbolos na frente dos testes indicam sucesso ou falha: uma marca de verificação para sucesso e um *x* para falha. Nenhum dos testes nesse script falhou, portanto não há nenhum script de erro nem uma saída com a stack trace.

Para os fãs do CoffeeScript, o Nodeunit tem suporte para aplicações desse tipo.

Outros frameworks de teste

Além do Nodeunit, discutido na seção anterior, há vários outros frameworks de testes disponíveis para desenvolvedores de Node. Algumas das ferramentas são mais simples de usar que outras, e cada uma delas tem suas próprias vantagens e desvantagens. A seguir, descreverei rapidamente três frameworks: Mocha, Jasmine e Vows.

Mocha

Instale o Mocha com o npm:

```
npm install mocha -g
```

O Mocha é considerado o sucessor de outro framework popular de testes, o Expresso.

Funciona tanto em navegadores quanto em aplicações Node. Ele permite testes assíncronos por meio da função `done`, embora a função possa ser omitida para testes síncronos. O Mocha pode ser usado com qualquer

biblioteca de asserção.

A seguir, vemos um exemplo de um teste do Mocha usando Assert:

```
assert = require('assert')
describe('MyTest', function() {
  describe('First', function() {
    it('sample test', function() {
      assert.equal('hello', 'hello');
    });
  });
});
```

Execute o teste com a linha de comando a seguir:

```
mocha testcase.js
```

O teste deverá resultar em sucesso:

```
MyTest
  First
    ✓ sample test

1 passing (15ms)
```

Vows

O Vows é um framework de testes de BDD (behavior-driven development, ou desenvolvimento orientado a comportamento) e tem uma vantagem sobre os demais frameworks: sua documentação é mais completa. Os testes são compostos de suítes que, por sua vez, são constituídos de batches (lotes) de testes executados sequencialmente. Um *batch* é composto de um ou mais contextos executados em paralelo e cada um é constituído de um *tópico* (topic). O teste no código é conhecido como *vow*. Um ponto em relação ao qual o Vows se orgulha de ser diferente dos demais frameworks de testes é o fato de ele oferecer uma separação clara entre o que está sendo testado (tópico) e o teste (vow).

Sei que esses são usos meio estranhos de palavras conhecidas; portanto, vamos observar um exemplo simples para ter uma melhor ideia do funcionamento dos testes no Vows. Porém, inicialmente devemos instalá-lo:

```
npm install vows
```

Para testar o Vows, usarei um módulo `circle` simples que devolve a área e a circunferência de um círculo. Como os valores são números de ponto flutuante e estou testando a igualdade, vou limitar os valores devolvidos a quatro casas decimais:

```
const PI = Math.PI;

exports.area = function (r) {
  return (PI * r * r).toFixed(4);
};

exports.circumference = function (r) {
  return (2 * PI * r).toFixed(4);
};
```

Na aplicação de teste do Vows, o objeto `circle` é o *tópico* e os métodos `area` e `circumference` são os *vows*. Ambos estão encapsulados como um *contexto do Vows*. A *suíte* é a aplicação de teste em geral, e o *batch* é a instância do teste (`circle` e os dois métodos). O Exemplo 11.2 mostra o teste completo.

Exemplo 11.2 – Aplicação de teste do Vows com um batch, um contexto, um tópico e dois vows

```
var vows = require('vows'),
  assert = require('assert');

var circle = require('./circle');

var suite = vows.describe('Test Circle');

suite.addBatch({
  'An instance of Circle': {
    topic: circle,
    'should be able to calculate circumference': function (topic) {
      assert.equal (topic.circumference(3.0), 18.8496);
    },
    'should be able to calculate area': function(topic) {
      assert.equal (topic.area(3.0), 28.2743);
    }
  }
}).run();
```

Executar a aplicação com o Node fará o teste ser executado por causa do acréscimo do método `run` no final do método addBatch:

```
node vowstest.js
```

O resultado deverá mostrar dois testes com sucesso:

```
.. ✓ OK . 2 honored (0.012s)
```

O tópico é sempre uma função assíncrona ou um valor. Em vez de usar `circle` como tópico, eu poderia ter referenciado diretamente os métodos do objeto como tópicos – com um pouco de ajuda de closures de função:

```
var vows = require('vows'),
  assert = require('assert');

var circle = require('./circle');

var suite = vows.describe('Test Circle');

suite.addBatch({
  'Testing Circle Circumference': {
    topic: function() { return circle.circumference;},
    'should be able to calculate circumference': function (topic) {
      assert.equal (topic(3.0), 18.8496);
    },
  },
  'Testing Circle Area': {
    topic: function() { return circle.area;},
    'should be able to calculate area': function(topic) {
      assert.equal (topic(3.0), 28.2743);
    }
  }
}).run();
```

Nessa versão do exemplo, cada contexto é o objeto que recebe um título: `Testing Circle Circumference` e `Testing Circle Area`. Em cada contexto, há um tópico e um vow.

Você pode incorporar vários batches, cada um com diversos contextos que, por sua vez, podem ter múltiplos tópicos e múltiplos vows.

Mantendo o Node executando

Você fará o melhor que puder com a sua aplicação. Vai testá-la de forma abrangente e adicionará um tratamento de erros para que os erros sejam administrados de forma elegante. Apesar disso, podem surgir complicações no caminho – situações não planejadas que podem causar falhas em sua aplicação. Se isso ocorrer, será necessário ter uma maneira

de garantir que sua aplicação possa iniciar novamente, mesmo que você não esteja por perto para reiniciá-la.

O Forever é uma ferramenta que serve exatamente para isso – ele garante que sua aplicação reiniciará caso haja falhas. É também uma forma de iniciar a sua aplicação como um daemon que seja persistente não só durante a sessão de terminal atual. O Forever pode ser usado a partir da linha de comando ou incorporado como parte da aplicação. Se for utilizá-lo a partir da linha de comando, você deverá instalá-lo globalmente:

```
npm install forever -g
```

Em vez de iniciar uma aplicação diretamente com o Node, inicie-a com o Forever:

```
forever start -a -l forever.log -o out.log -e err.log finalserver.js
```

Valores default são aceitos para duas opções: `minUpTime` (definido com 1000 ms) e `spinSleepTime` (definido com 1000 ms).

O comando anterior inicia um script *finalserver.js* e especifica os nomes para o log do Forever, o log de saída e o log de erro. Também instrui a aplicação a concatenar as entradas de log se os arquivos de log já existirem.

Se algo acontecer com o script fazendo-o falhar, o Forever irá reiniciá-lo. A ferramenta também garante que uma aplicação Node continue executando, mesmo que você feche a janela de terminal usada para iniciar a aplicação.

O Forever tem opções e ações. O valor `start` na linha de comando que acabamos de mostrar é um exemplo de ação. As ações disponíveis são:

`start`
: Inicia um script.

`stop`
: Termina um script.

`stopall`
: Termina todos os scripts.

`restart`
: Reinicia o script.

`restartall`
: Reinicia todos os scripts que estão executando com o Forever.

`cleanlogs`
: Apaga todas as entradas de log.

`logs`
: Lista todos os arquivos de log de todos os processos do Forever.

`list`
: Lista todos os scripts em execução.

`config`
: Lista as configurações de usuário.

`set <chave> <valor>`
: Define chave e valor na configuração.

`clear <chave>`
: Limpa chave e valor da configuração.

`logs <script|índice>`
: Faz tail dos logs para <script|índice>

`columns add <col>`
: Adiciona uma coluna à lista de saída do Forever.

`columns rm <col>`
: Remove uma coluna da lista de saída do Forever.

`columns set <cols>`
: Define todas as colunas para a lista de saída do Forever.

Podemos ver um exemplo da saída de `list` a seguir, após *httpserver.js* ter sido iniciado como um daemon do Forever:

```
info: Forever processes running
data: uid command script forever pid id
logfile uptime
data: [0] _gEN /usr/bin/nodejs serverfinal.js 10216 10225
/home/name/.forever/forever.log STOPPED
```

Liste os arquivos de log com a ação `logs`:

```
info: Logs for running Forever processes
data: script logfile
data: [0] serverfinal.js /home/name/.forever/forever.log
```

Há também um número significativo de opções, incluindo as configurações dos arquivos de log que acabamos de mostrar, bem como para executar o script (`-s` ou `--silent`), ativar a verbosidade do Forever (`-v` ou `--verbose`), definir o diretório-fonte do script (`--sourceDir`) e outros; você poderá ver todos eles se digitar:

```
forever --help
```

Você pode incorporar o uso do Forever diretamente em seu código usando o seu módulo associado, o `forever-monitor`, conforme mostrado na documentação do módulo:

```
var forever = require('forever-monitor');

  var child = new (forever.Monitor)('serverfinal.js', {
    max: 3,
    silent: true,
    args: []
  });

  child.on('exit', function () {
    console.log('serverfinal.js has exited after 3 restarts');
  });

  child.start();
```

Além disso, o Forever pode ser usado com o Nodemon, não só para reiniciar a aplicação caso ela falhe de modo inesperado, mas também para garantir que ela seja atualizada se houver uma mudança no código-fonte.

Instale o Nodemon globalmente:

```
npm install -g nodemon
```

O Nodemon encapsula a sua aplicação. Em vez de usar o Node para iniciar a aplicação, utilize o Nodemon:

```
nodemon app.js
```

O Nodemon monitora silenciosamente o diretório (e quaisquer outros diretórios contidos) em que você executou a aplicação, verificando se houve mudanças nos arquivos. Se uma mudança for detectada, o Nodemon reiniciará a aplicação de modo que as mudanças recentes sejam consideradas.

Você pode passar parâmetros para a aplicação:

```
nodemon app.js param1 param2
```

Também pode usar o módulo com o CoffeeScript:

```
nodemon someapp.coffee
```

Se quiser que o Nodemon monitore algum diretório que não seja o atual, utilize a flag `--watch`:

```
nodemon --watch dir1 --watch libs app.js
```

Há outras flags documentadas com o módulo (*https://github.com/remy/nodemon/*).

Para usar o Nodemon com o Forever, encapsule-o no Forever e especifique a opção `--exitcrash` para garantir que, se a aplicação falhar, o Nodemon sairá de forma limpa e passará o controle para o Forever:

```
forever start nodemon --exitcrash serverfinal.js
```

Se você vir um erro informando que o Forever não encontrou o Nodemon, utilize o path completo:

```
forever start /usr/bin/nodemon --exitcrash serverfinal.js
```

Se a aplicação falhar, o Forever iniciará o Nodemon que, por sua vez, iniciará o script Node, garantindo que este não só execute de forma atualizada caso o código-fonte seja alterado, como também que uma falha inesperada não deixará sua aplicação permanentemente offline.

Benchmark e testes de carga com o Apache Bench

Uma aplicação robusta, que atenda a todas as necessidades de usuário, terá uma vida curta se o seu desempenho for atroz. Devemos ter a capacidade de fazer *testes de desempenho* em nossas aplicações Node,

especialmente quando fazemos ajustes como parte do processo para melhoria do desempenho. Não podemos simplesmente ajustar a aplicação, colocá-la em ambiente de produção e deixar que os nossos usuários identifiquem os problemas de desempenho.

Os testes de desempenho são compostos de testes de benchmark e testes de carga. Os *testes de benchmark*, também conhecidos como *testes de comparação*, consistem em executar várias versões ou variantes de uma aplicação e então determinar qual delas é a melhor. É uma ferramenta eficaz quando ajustamos uma aplicação a fim de melhorar a sua eficiência e escalabilidade. Você deve criar um teste padronizado, executá-lo nas variantes e então analisar os resultados.

Por outro lado, o teste de carga consiste em fazer um teste de estresse em sua aplicação. Você estará tentando ver em que ponto a sua aplicação começará a falhar ou a ter problemas como consequência de muita demanda de recursos ou do excesso de usuários concorrentes. Basicamente, você quer executar a aplicação até que ela falhe. A falha é um sucesso nos testes de carga.

Há ferramentas que lidam com os dois tipos de testes de desempenho, e uma ferramenta popular para isso é o ApacheBench. Ela é popular porque está disponível por padrão em qualquer servidor que tiver o Apache instalado – e poucos servidores não o têm. É também uma pequena ferramenta de testes fácil de usar e eficaz. Quando estava tentando determinar se seria melhor criar uma conexão estática com o banco de dados para reutilizá-la ou criar uma conexão e descartá-la a cada uso, usei o ApacheBench para executar os testes.

O ApacheBench é comumente chamado de ab, e usarei esse nome a partir de agora. `O ab é uma` ferramenta de linha `de comando que nos permite` especificar o número de vezes que uma aplicação é executada e por quantos usuários concorrentes. Se quiser emular 20 usuários concorrentes acessando uma aplicação web um total de 100 vezes, usaríamos um comando como este:

```
ab -n 100 -c 20 http://burningbird.net/
```

É importante especificar a barra final, pois o ab espera um URL completo,

incluindo o path.

A ferramenta ab oferece uma saída muito rica em informações. Um exemplo é a saída a seguir (não inclui a identificação da ferramenta) de um teste:

```
Benchmarking burningbird.net (be patient).....done

Server Software: Apache/2.4.7
Server Hostname: burningbird.net
Server Port: 80

Document Path: /
Document Length: 36683 bytes

Concurrency Level: 20
Time taken for tests: 5.489 seconds
Complete requests: 100
Failed requests: 0
Total transferred: 3695600 bytes
HTML transferred: 3668300 bytes
Requests per second: 18.22 [#/sec] (mean)
Time per request: 1097.787 [ms] (mean)
Time per request: 54.889 [ms] (mean, across all concurrent requests)
Transfer rate: 657.50 [Kbytes/sec] received

Connection Times (ms)
              min mean[+/-sd] median max
Connect: 0 1 2.3 0 7
Processing: 555 1049 196.9 1078 1455
Waiting: 53 421 170.8 404 870
Total: 559 1050 197.0 1080 1462

Percentage of the requests served within a certain time (ms)
  50% 1080
  66% 1142
  75% 1198
  80% 1214
  90% 1341
  95% 1392
  98% 1415
  99% 1462
 100% 1462 (longest request)
```

As linhas em que estamos mais interessados são aquelas que têm a ver com a duração de cada teste e a distribuição cumulativa no final do teste

(baseada em porcentagens). De acordo com essa saída, o tempo médio por requisição (o primeiro valor identificado com `Time per request`) foi de 1097,787 milissegundos. Esse é o tempo que o usuário em média poderia esperar por uma resposta. A segunda linha está relacionada com o throughput e, provavelmente, não é tão útil quanto a primeira.

A distribuição cumulativa oferece uma boa visão da porcentagem de requisições tratada em determinado intervalo de tempo. Novamente, esses dados informam o que poderíamos esperar para um usuário em média: tempos de resposta entre 1.080 e 1.462 milissegundos, com a grande maioria das respostas tratadas em 1.392 milissegundos ou menos.

O último valor que estamos analisando é o de requisições por segundo: nesse caso, 18,22. Esse valor, de certo modo, é capaz de prever quão bem a aplicação vai escalar, pois nos dá uma ideia do máximo de requisições por segundo – isto é, o limite superior para acessos à aplicação. No entanto, você precisará executar o teste em momentos diferentes, com cargas secundárias distintas, especialmente se estiver executando o teste em um sistema utilizado para outros propósitos.

A aplicação Loadtest

Você também pode usar a aplicação Loadtest para fazer testes de carga:

```
npm install -g loadtest
```

A vantagem em relação ao Apache Bench é que podemos definir a quantidade de requisições, bem como de usuários:

```
loadtest [-n requests] [-c concurrency] [-k] URL
```

CAPÍTULO 12

Node em novos ambientes

O Node se expandiu para muitos ambientes diferentes, para além do servidor básico em Linux, OS X e Windows.

O Node oferece uma opção para usar JavaScript com microcomputadores e microcontroladores como o Raspberry Pi e o Arduino. A Samsung está planejando integrar o Node em sua visão de IoT (Internet of Things, ou Internet das Coisas), embora a tecnologia de IoT que ela adquiriu, a SmartThings, seja baseada em uma variante de Java (Groovy). Além disso, a Microsoft adotou o Node e, atualmente, está trabalhando para expandi-lo, oferecendo uma variação dele executando-a com a sua própria engine, o Chakra.

São as muitas faces do Node que o fazem ser, ao mesmo tempo, empolgante e divertido.

O Node em ambientes móveis

O Node também encontrou o seu caminho no mundo dos dispositivos móveis, mas tentar espremer uma seção sobre esse tópico neste livro estava além de minha capacidade de compactá-lo a um nível mínimo. Como alternativa, indicarei ao leitor um livro sobre o assunto: *Learning Node.js for Mobile Application Development* (Packt, 2015, de Stefan Buttigieg e Milorad Jevdjenic.

IoT da Samsung e GPIO

A Samsung criou uma variante do Node chamada IoT.js, assim como uma versão JavaScript para tecnologias IoT, chamada JerryScript. Com base na documentação, o principal motivo para as novas variantes é desenvolver ferramentas e tecnologias que funcionem em dispositivos com menos memória que os ambientes tradicionais que usam JavaScript/Node.

Em um gráfico que acompanha uma apresentação de um funcionário da

Samsung (*http://bit.ly/1YklQea*), uma implementação completa do JerryScript foi apresentada, com um tamanho binário de 200 KB e um uso de memória de 16 KB a 64 KB – isso em comparação com o V8 que exige um tamanho binário de 10 MB e um uso de memória de 8 MB. Quando trabalhamos com dispositivos IoT, qualquer fragmento de espaço ou de memória é importante.

Pensando na IoT.js, em sua documentação (*https://github.com/Samsung/iotjs/wiki/IoT.js-API-Reference*), você verá que ela oferece suporte para um subconjunto de módulos básicos do Node como Buffer, HTTP, Net e File System. Considerando o seu uso para propósitos mais específicos, a ausência de suporte para módulos como Crypto é compreensível. Você saberá que está no mundo da IoT quando vir que a IoT.js inclui também um novo módulo básico: o GPIO. Esse módulo representa a interface da aplicação com o hardware físico e constitui a ponte entre a aplicação e o dispositivo.

GPIO é o acrônimo para general-purpose input/output (entrada/saída de propósito geral). A GPIO representa um pino em um circuito integrado, que pode ser tanto de entrada quanto de saída e cujo comportamento é controlado pelas aplicações que criamos. Os pinos da GPIO oferecem uma interface para o dispositivo. Como entradas, eles podem receber informações dos dispositivos, como temperatura ou sensores de movimento; como saídas, são capazes de controlar luzes, telas touch, motores, dispositivos de rotação, e assim por diante.

Em um dispositivo como o Raspberry Pi (discutido na seção “Node para microcontroladores e microcomputadores”), há um painel de pinos em um dos lados, no qual a maior parte são pinos GPIO, em meio a pinos terra e de alimentação. A Figura 12.1 mostra uma foto dos pinos e, abaixo dela, um *diagrama de pinagem* que mostra os pinos de GPIO, alimentação e terra de um Raspberry Pi 2 Modelo B.

Como você pode notar, o número dos pinos no diagrama de pinagem não reflete a sua localização física propriamente dita na placa. O número que aparece no rótulo do pino é o *número GPIO*. Algumas APIs, incluindo a IoT.js da Samsung, esperam o número GPIO quando pedem o número de

um pino.

Para usar a IoT.js da Samsung, inicialize o objeto GPIO e, em seguida, chame uma das funções, como `gpio.setPen()`, que aceita o número de um pino como primeiro parâmetro, a direção (isto é, `'in'` para entrada, `'out'` para saída e `'none'` para disponibilizar), um modo opcional e uma callback. Para enviar dados ao pino, utilize a função `gpio.writePin()`, fornecendo-lhe o número do pino, um valor booleano e uma callback.
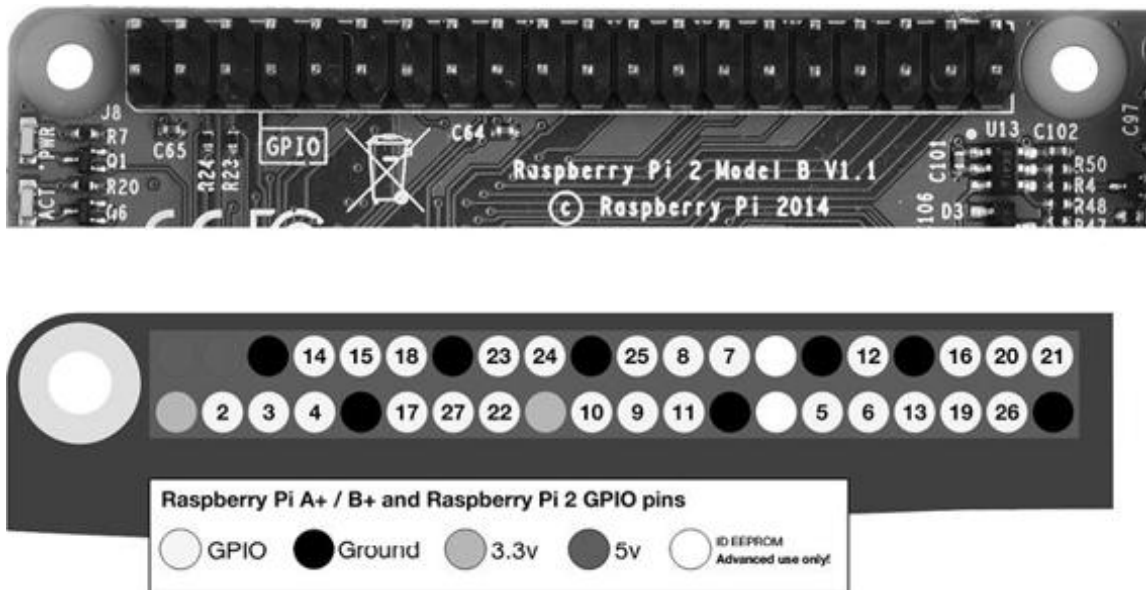
Figura 12.1 – Pinos do Raspberry Pi 2 e diagrama de pinagem associado; cortesia da Raspberry Pi Foundation (http://bit.ly/1Omurrd), usado com licença CC CC-BY-SA.

A IoT.js da Samsung (*https://samsung.github.io/iotjs/*) definitivamente é um trabalho em andamento. Além do mais, parece que ela apresenta conflitos com outro trabalho da Samsung conhecido como SAMIO (*https://developer.samsungsami.io*), o qual define uma plataforma para troca de dados que, entre outras tarefas, possibilita a comunicação entre um Arduino e um Raspberry Pi na monitoração de um sensor de incêndio (veja o tutorial associado em *https://developer.samsungsami.io/sami/tutoriais/your-first-iot-device.html*) – e incorpora o uso do Node. No entanto, tudo está em desenvolvimento ativo, portanto acho que devemos nos manter sintonizados.

Raspberry Pi e Arduino

Veremos o Raspberry Pi e o Arduino com mais detalhes na seção "Node para microcontroladores e microcomputadores".

Windows com Chakra Node

Em 19 de janeiro de 2016, a Microsoft criou uma PR (pull request, em *https://blogs.windows.com/msedgedev/2016/01/19/nodejs-chakracore-mainline/*) para permitir que o Node executasse com a engine ChakraCore (JavaScript) da Microsoft. A empresa criou um complemento para V8 que implementa a maioria das APIs essenciais a ele, permitindo que o Node execute de forma transparente no ChakraCore.

A ideia de o Node executar em algo que não seja o V8 é, ao mesmo tempo, interessante e, em minha opinião, atraente. Embora pareça que o V8 esteja direcionando o desenvolvimento do Node, pelo menos das versões Node Current, tecnicamente não há nenhum requisito absoluto de que o Node seja construído sobre o V8. Afinal de contas, as atualizações nas versões Current estão mais voltadas para as melhorias na API do Node e a incorporação de inovações da ECMAScript.

A Microsoft disponibilizou o ChakraCore como código aberto, o que é um primeiro passo necessário. Ele oferece suporte de engine JavaScript para o novo navegador da empresa, o Edge, e a empresa argumenta que essa engine é superior ao V8.

Enquanto o debate e os testes sobre a PR continuam em andamento, você pode testar o Node no ChakraCore, desde que tenha um computador com Windows, Python (2.6 ou 2.7) e o Visual Studio (por exemplo, o Visual Studio 2015 Community, disponível para download gratuito em *https://www.visualstudio.com/en-us/products/visual-studio-community-vs.aspx*). Você também pode fazer download de um binário pronto. A Microsoft disponibiliza binários para ARM (para Raspberry Pi), assim como para as arquiteturas x86 e x64, mais tradicionais. Quando fiz a instalação em meu computador Windows, o mesmo tipo de janela de Comando (Command) usado no Node sem o ChakraCore foi criado. O melhor de tudo é que ele pode ser instalado no mesmo computador em que está o Node, e os dois podem ser usados lado a lado.

Testei vários exemplos do livro e não me deparei com nenhum problema. Também testei um exemplo que reflete uma alteração recente no Node – ela permite que a função `child_process.spawn()` especifique uma opção de shell (discutida no Capítulo 8). O exemplo não funcionou com a versão 4.x LTS, mas funciona com a versão Current mais recente (6.0.0, quando este livro foi escrito). E funcionou com a build binária do Node baseada no ChakraCore, apesar de a nova versão Current ter sido lançada alguns dias antes. Portanto, a Microsoft está usando a versão Current mais recente da API do Node em sua build do binário do Node/ChakraCore.

Mais interessante ainda é o teste que fiz do exemplo com reflect/proxy da ES6 a seguir:

```
'use strict'

// exemplo, cortesia do Dr. Axel Rauschmayer
// http://www.2ality.com/2014/12/es6-proxies.html

let target = {};
  let handler = {
    get(target, propKey, receiver) {
      console.log('get ' + propKey);
      return 123;
    }
  };

let proxy = new Proxy(target, handler);

console.log(proxy.foo);

proxy.bar = 'abc';
console.log(target.bar);
```

O exemplo funcionou com o ChakraCore Node, mas não com o V8 Node, nem mesmo com a versão mais recente. Outra vantagem defendida pelos desenvolvedores que usam ChakraCore é que ele apresenta uma implementação superior das novas melhorias da ECMAScript.

O exemplo não funcionou na V5 Stable, mas funcionou com a V6 Current Node mais recente.

Atualmente, o ChakraCore só funciona no Windows, e um binário para Raspberry Pi é disponibilizado. Também prometeram que haverá uma versão para Linux em breve.

Node para microcontroladores e microcomputadores

O Node encontrou um lar muito confortável junto aos microcontroladores, como o Arduino, e aos microcomputadores, como o Raspberry Pi.

Uso *microcomputador* nas discussões sobre o Raspberry Pi, mas na verdade ele é um computador totalmente funcional – apesar de ser pequeno. Você pode instalar um sistema operacional nele, como o Windows 10 ou o Linux, conectar um teclado, um mouse e um monitor e executar muitas aplicações, incluindo um navegador web, jogos ou aplicações de escritório. O Arduino, por outro lado, é usado para tarefas repetitivas. Em vez de conectar um teclado ou um monitor diretamente ao dispositivo, você deve conectar o dispositivo ao seu computador e usar uma aplicação associada para construir e carregar um programa no dispositivo. É um *computador embarcado*, se comparado com um computador (como o Raspberry Pi).

Usando o Raspberry Pi e o Arduino juntos

Você não precisa escolher entre o Raspberry Pi e o Arduino; eles podem ser usados em conjunto. O Pi será o cérebro, e o Arduino, o músculo.

O Arduino e o Raspberry Pi são populares em suas respectivas categorias, mas são apenas um subconjunto dos dispositivos disponíveis, muitos dos quais são compatíveis com o Arduino. Há até um dispositivo que pode ser vestido (LilyPad, *https://www.arduino.cc/en/Main/ArduinoBoardLilyPad*).

Se a IoT e o desenvolvimento para dispositivos conectados forem novidade, recomendo que você comece com o Arduino Uno (*https://www.arduino.cc/*) e, em seguida, experimente usar o Raspberry Pi 2 (*https://www.raspberrypi.org/*). Existem kits disponíveis em sites online como AdaFruit, SparkFun, Cana Kit, Amazon, Maker Shed e outros, assim como nos sites dos próprios fabricantes das placas. Os kits incluem tudo que é necessário para começar a trabalhar com as placas e não são caros (menos de 100 dólares). Livros de projeto acompanham os kits, além dos componentes necessários para vários deles.

Nesta seção, demonstrarei a criação de uma aplicação Hello World para

esses tipos de dispositivo. A aplicação é constituída de um programa que acessa uma lâmpada LED em um pino GPIO e a faz piscar. Demonstrarei a aplicação em um Arduino Uno e em um Raspberry Pi 2.

O novo Raspberry Pi 3

Quando estava terminando este livro, o Raspberry Pi 3 havia sido lançado. Deverá ser fácil converter o exemplo com Raspberry Pi, mais adiante no livro, para o dispositivo mais recente. O melhor de tudo é que o Raspberry Pi 3 tem WiFi integrado.

Inicialmente, porém, você não poderá ir longe no mundo dos dispositivos conectados sem conhecer um pouco de eletrônica e a maravilhosa ferramenta Fritzing.

Fritzing

O Fritzing é um software de código aberto que oferece às pessoas as ferramentas para fazer o protótipo de um design e então fazer o design ser fisicamente criado por meio do Fritzing Fab. O software está disponível gratuitamente (mas recomendo fazer uma doação se achar que a ferramenta é útil). Se vir diagramas gráficos de projetos para Arduino e Raspberry Pi, eles serão invariavelmente feitos com a aplicação Fritzing.

Faça download da aplicação

A aplicação Fritzing pode ser baixada a partir do site do Fritzing (*http://fritzing.org/home/*). Não é uma aplicação pequena, o que não é nenhuma surpresa considerando o número de componentes gráficos necessários à aplicação. Você pode fazer download de uma versão para Windows, OS X ou Linux.

Quando você abrir a ferramenta e criar um novo *sketch* (esboço), como é chamado o diagrama de design, uma placa de ensaio (breadboard ou protoboard) será adicionada automaticamente ao sketch. Uma *placa de ensaio* é uma forma fácil de criar o protótipo de um projeto elétrico. Você não precisará soldar nenhum componente; tudo que você deverá fazer é conectar uma extremidade do fio ao pino da placa e a outra extremidade à placa de ensaio. Embaixo da parte externa de plástico da placa de ensaio, há faixas de metal condutor: faixas horizontais longas embaixo das *trilhas de alimentação* se a placa as tiver (delimitadas pelas linhas vermelha e azul, em destaque na Figura 12.2) e faixas mais curtas verticais embaixo da *faixa de terminais* em cada coluna (exibida na Figura 12.3). Você deve

então adicionar todos os componentes do projeto à placa de ensaio.
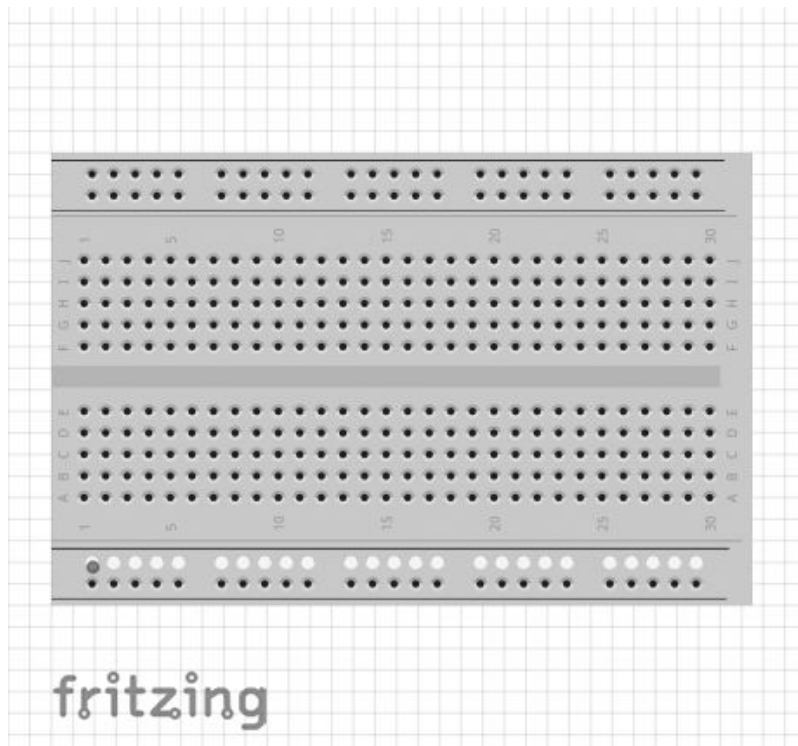
Figura 12.2 – Diagrama da placa de ensaio destacando a conexão com a trilha de alimentação.
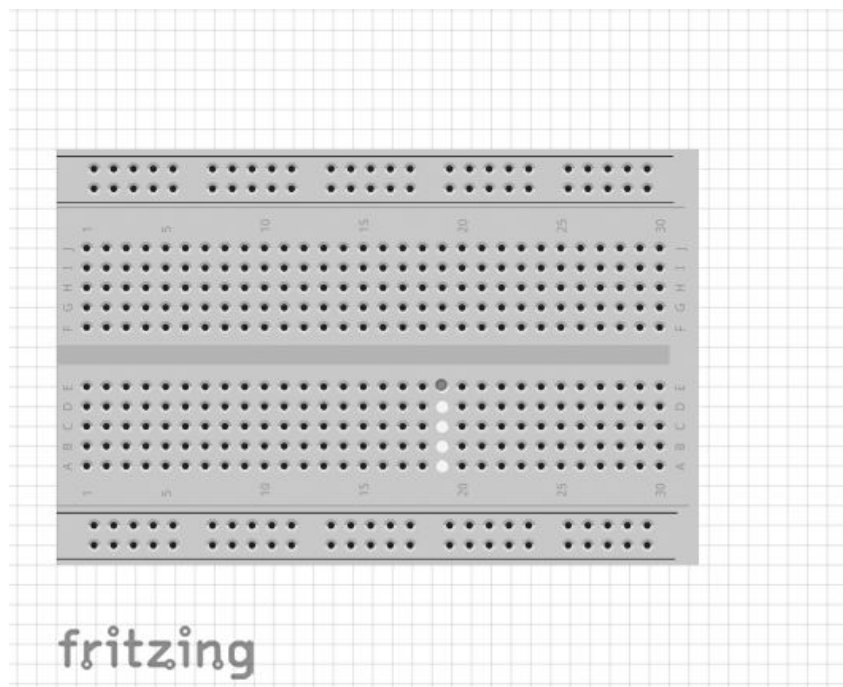
Figura 12.3 – Diagrama da placa de ensaio destacando a faixa de terminais.

Anatomia de uma placa de ensaio

Para ter uma visão mais detalhada da placa de ensaio, sua história e como ela funciona, recomendo o tutorial “How to Use a Breadboard” (Como usar uma placa de ensaio, *https://learn.sparkfun.com/tutorials/how-to-use-a-breadboard*) da SparkFun.

Você deve arrastar e soltar os componentes que vai usar a partir das famílias de componentes à direita na ferramenta. Na Figura 12.4, podemos ver o sketch que fiz para o exemplo com o LED piscante no Arduino na próxima seção. Ele é composto de um Arduino Uno, arrastado do painel à direita, uma placa de ensaio com a metade do tamanho, dois fios de conexão, um resistor e um LED da mesma lista de componentes, com a cor do LED alterada de vermelho para azul. A descrição dos componentes é exibida na parte inferior à direita. Você pode editar manualmente as informações sobre todos os componentes nesta seção, incluindo a placa de ensaio (modificada de inteira para metade).

Para criar os fios, clique e arraste do pino do Arduino para o local em que você quer que o fio termine na placa de ensaio. A ferramenta adicionará automaticamente o fio. Adicione cada componente arrastando-o a partir do menu de componentes para o sketch. Para garantir que o circuito seja preciso e completo, você também poderá conferir a visão esquemática do projeto, como mostra a Figura 12.5.
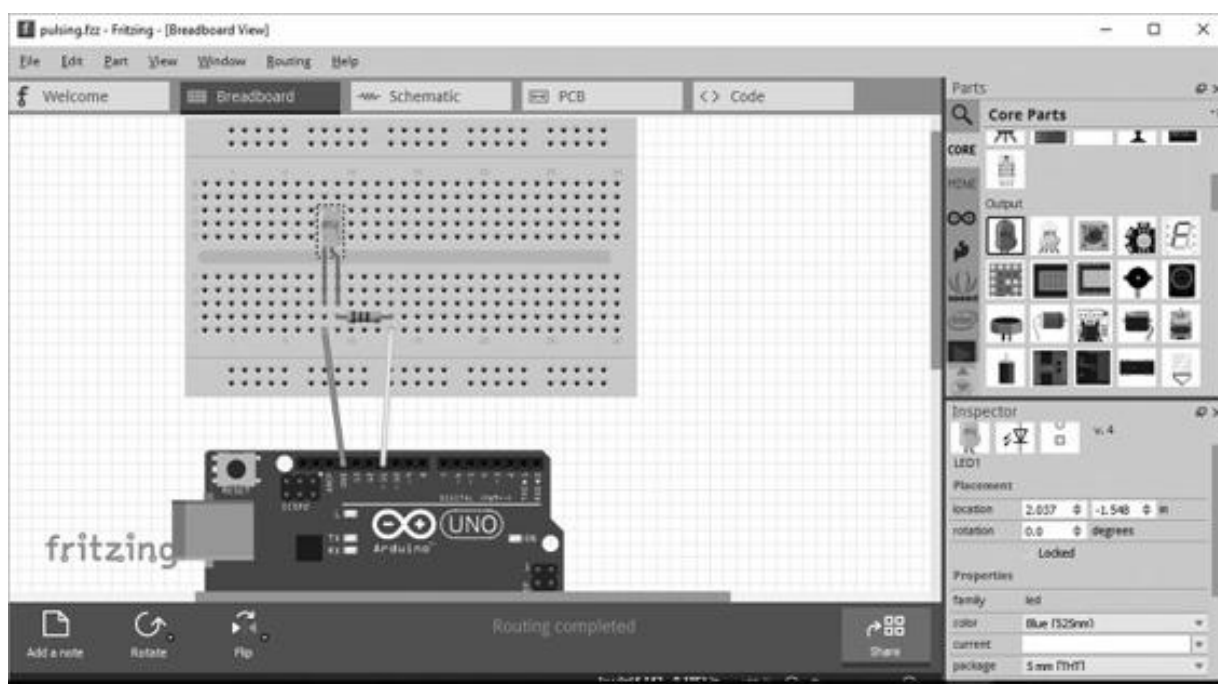
Figura 12.4 – Sketch do Fritzing para o projeto de LED piscante com o Arduino.
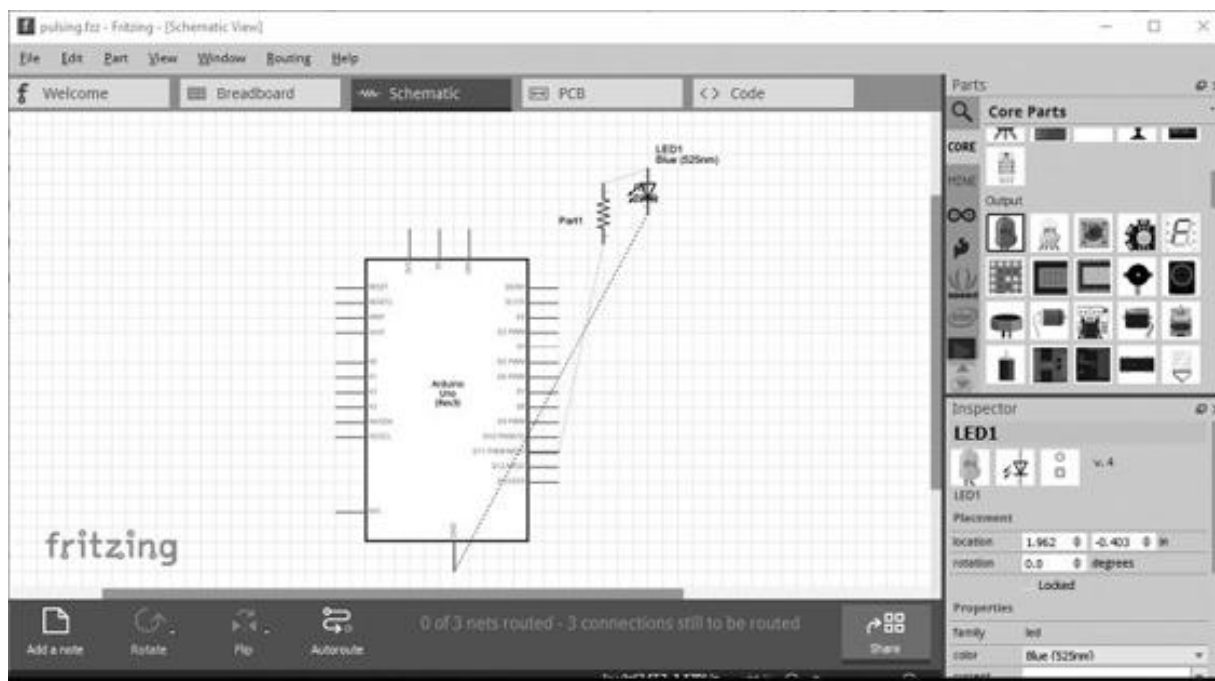
Figura 12.5 – Diagrama esquemático para o projeto do LED piscante com o Arduino.

O LED tem duas pernas, chamadas de *terminais* (leads), uma mais longa que a outra. No diagrama, o terminal mais longo está dobrado. Este é o terminal positivo (ânodo), enquanto o terminal mais curto é o negativo (cátodo). No sketch, o terminal negativo está conectado ao terra do Arduino, enquanto o terminal positivo está conectado ao pino 11.

Um fio é ligado do pino terra (GND) à placa. O LED é colocado na placa com o terminal mais curto (cátodo) posicionado na mesma coluna que o fio do terra. Um resistor é colocado na placa, com um terminal na mesma coluna que o terminal ânodo do LED. Por fim, um fio que conecta o outro terminal do resistor ao pino 11 do Arduino é adicionado, completando o circuito.

O resistor é o novo componente nesse sketch. Os resistores fazem o que o seu nome implica: resistem à corrente elétrica. Se um resistor não fosse adicionado ao circuito (a sequência completa é um circuito), o LED poderia tentar consumir muita eletricidade, podendo possivelmente

danificar o pino GPIO e a placa.

A resistência é medida em ohms (Ω) e o resistor no sketch é de 220 ohms. Você pode dizer de quantos ohms é um resistor lendo as suas faixas. Cores diferentes em faixas distintas (e um número de faixas diferente) definem o tipo do resistor incluído no circuito.

Informações sobre eletrônica

Como sabemos qual resistor deve ser usado? É por meio de algo chamado Lei de Ohm. Sim, tudo aquilo que você aprendeu na escola poderá finalmente ser usado. Para saber mais sobre a Lei de Ohm e seleção de resistores, consulte “Calculating correct resistor value to protect Arduino pin” (Calculando o valor correto do resistor para proteger os pinos do Arduino, *http://bit.ly/1svAlvP*) e “Do I really need resistors when controlling LEDs with Arduino?” (Realmente preciso de resistores para controlar LEDs no Arduino?, *http:/bit.ly/1Veh8Qk*).

O LED é *polarizado*, o que significa que ele tem uma direção. O modo como ele é posicionado na placa (e no circuito) é importante, com o cátodo ligado ao terra. No entanto, o resistor não é polarizado, portanto pode ser colocado em qualquer direção na placa. Além do mais, o resistor não precisa ser colocado depois do LED; conforme mostra a Figura 12.6, ele pode ser colocado antes dele. Como este é um circuito simples, tudo que importa é que o circuito tenha um resistor. A Figura 12.6 mostra o sketch do Fritzing para o projeto com Raspberry Pi discutido na seção “Node e Raspberry Pi 2”, com o resistor posicionado antes do LED.
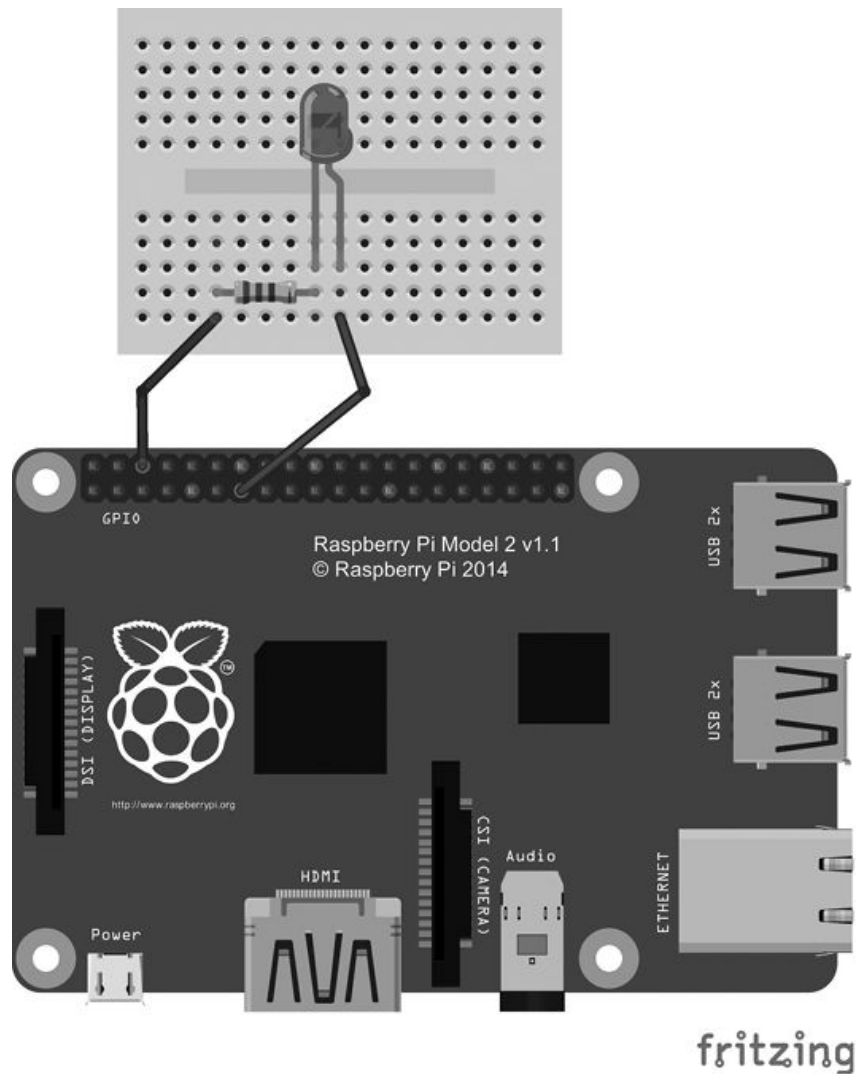
Figura 12.6 – Sketch no Fritzing para o projeto de LED com Raspberry Pi.

Essa foi uma introdução extremamente rápida para apenas alguns dos componentes de um circuito, mas é tudo de que você precisará para trabalhar com os exemplos nas duas próximas seções.

Node e Arduino

Para programar um Arduino Uno, você deve instalar o software do Arduino (*https://www.arduino.cc/en/Main/Software*) em seu computador. Usei a versão 1.7.8 no exemplo, em um computador com Windows 10. Há versões para OS X e para Linux. Os sites do Arduino oferecem instruções de instalação excelentes e detalhadas (*https://www.arduino.cc/en/Guide/HomePage*). Feita a instalação, conecte o

Arduino ao PC com um cabo USB e anote a porta serial para se conectar com a placa (COM3 no meu caso).

Em seguida, você deverá fazer upload do *firmata* no Arduino para usar o Node. O firmata implementa o protocolo para comunicação com o microcontrolador usando um software no computador. Na aplicação Arduino, selecione **File→Examples→Firmata→StandardFirmata** (Arquivo→Exemplos→Firmata→StandardFirmata). A Figura 12.7 mostra o firmata carregado na aplicação. Há uma seta que aponta para a direita na parte superior da janela. Clique nela para fazer upload do firmata no Arduino.
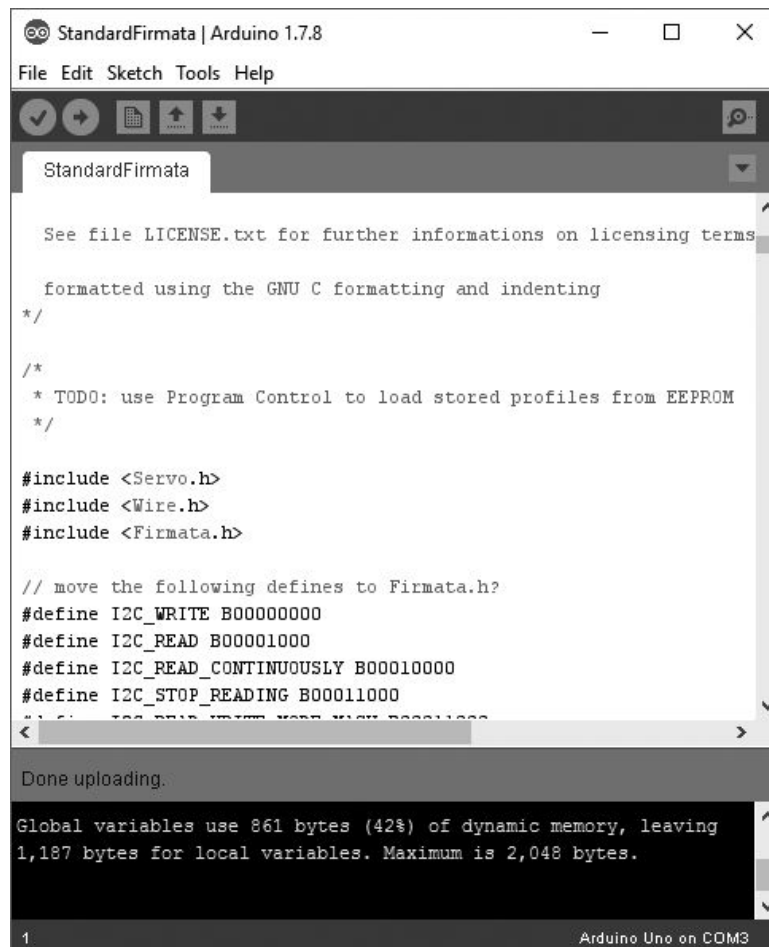
Figura 12.7 – O firmata, que deve ser carregado no Arduino para o Node funcionar.

Node e JavaScript não são a forma default nem a maneira mais comum de controlar o Arduino ou o Raspberry Pi (Python é a opção popular). No

entanto, há um framework Node chamado Johnny-Five (*http://johnny-five.io/*) que oferece uma maneira de programar esses dispositivos. Instale-o no Node usando o seguinte comando:

```
npm install johnny-five
```

O site do Johnny-Five oferece não só uma descrição da API abrangente que você pode usar para controlar a placa, como também tem diversos exemplos, incluindo a aplicação piscante "Hello World". De modo diferente do exemplo fornecido no site do Johnny-Five, que realmente conecta um LED diretamente aos pinos da placa Arduino Uno, faremos piscar o LED embutido na placa. Podemos acessar esse LED usando o pino de número 13.

Eis o código para fazer o LED embutido piscar (ou um LED externo conectado ao pino 13):

```
var five = require("johnny-five");
var board = new five.Board();

board.on("ready", function() {
  var led = new five.Led(13);
  led.blink(500);
});
```

A aplicação carrega o módulo Johnny-Five e cria uma nova placa, que representa o Arduino. Quando a placa estiver pronta, a aplicação cria um novo LED usando o pino de número 13. Observe que esse não é o mesmo número da GPIO; o Johnny-Five utiliza um sistema de numeração físico que reflete as localizações dos pinos marcados na placa. Depois que o objeto LED é criado, sua função `blink()` é chamada. O resultado está na Figura 12.8, com uma seta apontando para o LED.
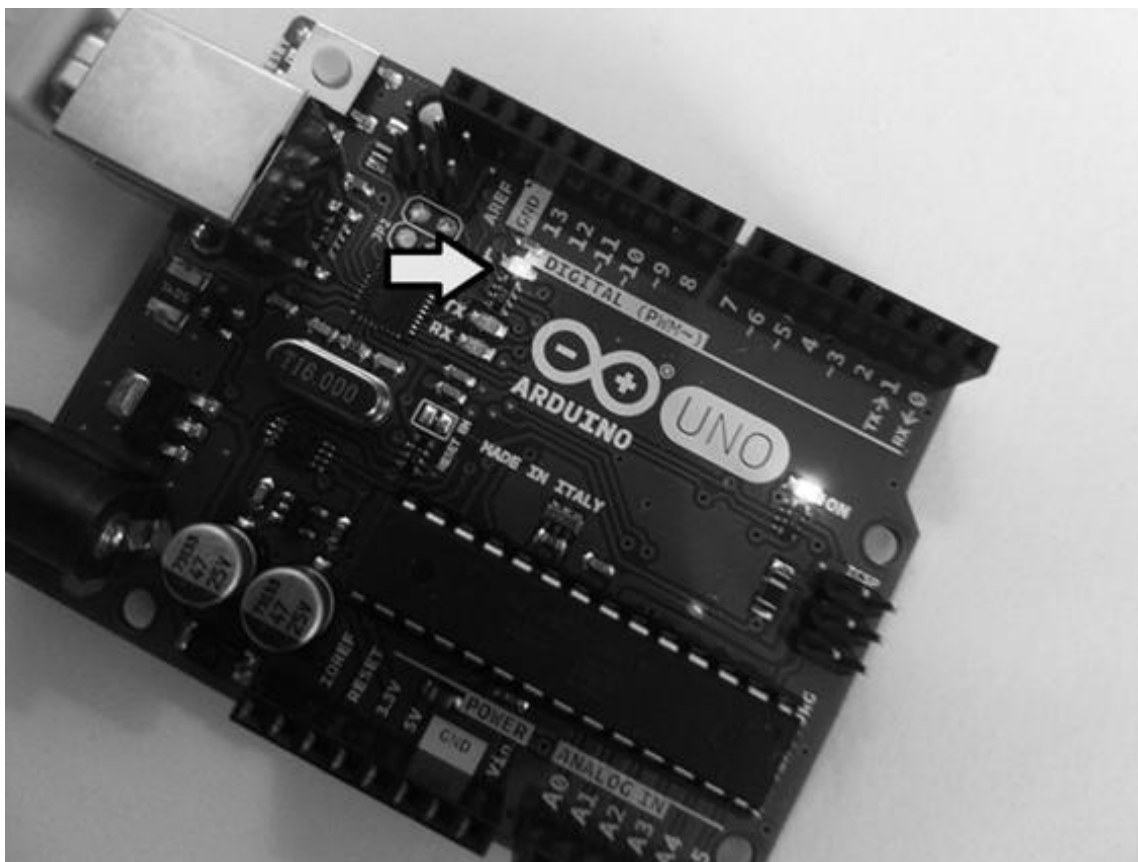
Figura 12.8 – Piscando o LED embutido ligado ao pino 13 no Arduino Uno.

A aplicação abre um REPL. Para encerrá-la, digite `.exit`. O LED continuará a piscar até você encerrar a aplicação; se o LED estiver aceso quando a aplicação for encerrada, ele permanecerá assim até você desligar o dispositivo.

Segurança, segurança, segurança

Tudo na IoT é diversão e jogos até você queimar a sua placa. Ou o seu computador.

Falando seriamente, você pode trabalhar de modo fácil e seguro com a maioria das aplicações de IoT para pequenos dispositivos, desde que siga as instruções. É por esse motivo que é importante conferir sempre os diagramas quando estiver colocando os componentes e usar os componentes corretos.

Também é importante manter o seu espaço de trabalho limpo e organizado, utilizar uma boa iluminação, garantir que você não ande em um tapete de nylon usando meias antes de tocar em qualquer componente eletrônico (você deve estar aterrado), desconectar as placas antes de alterar os componentes e manter o gato curioso e o cão hiperativo (ou crianças pequenas) fora do cômodo em que você estiver trabalhando. Também não recomendo deixar uma xícara de café ao lado de sua placa.

Você também deve guardar os componentes eletrônicos quando terminar de trabalhar com

eles se tiver crianças pequenas – há poucos componentes além das placas que não podem ser engolidos por uma criança, e os LEDs parecem particularmente comestíveis. Os projetos são uma excelente experiência de aprendizagem para crianças mais velhas, mas use o bom senso em relação à idade que as crianças devem ter para trabalhar de modo independente nos projetos.

Eu uso óculos, mas se você não os usa, deverá pensar seriamente em obter óculos de proteção para trabalhar.

Alguns dispositivos de segurança para os eletrônicos estão disponíveis. A placa Arduino Uno tem reguladores na placa que devem evitar que esse pequeno dispositivo bonitinho queime o seu computador. No entanto, a própria placa é suscetível a danos. As placas não são caras, mas não a ponto de não causar sofrimento se elas queimarem ou se seus pinos forem danificados com frequência.

Obtenha mais informações em Electronics no StackExchange (*http://bit.ly/1qzshP2*).

O LED piscante é divertido, mas agora que temos esse novo brinquedo interessante, vamos realmente explorá-lo. O objeto LED tem suporte para várias funções robustas interessantes, como `pulse()` e `fadeIn()`. No entanto, essas funções exigem um pino PWM (pulse width modulation, ou modulação por largura de pulso), também conhecido como *saída analógica*. O pino 13 não é PWM. Entretanto, o pino 11 é, e você pode identificar isso pelo caractere de til (~) que antecede a sua identificação na placa (~11).

Primeiro, desligue a placa. Em seguida, reúna a placa de ensaio, o LED, o resistor de 220 ohms e dois fios para conexão. A maioria dos kits deverá ter todos esses componentes.

Vire a placa de modo que o rótulo "Arduino" esteja voltado para cima. Ao longo da parte superior, há uma fila de pinos. Ligue um fio de conexão no pino terra, identificado como GND. Conecte um segundo fio no pino identificado com ~11. Em uma placa de ensaio ao lado do Arduino, siga o sketch do Fritzing mostrado na Figura 12.9 para posicionar o restante dos componentes na placa. Quando terminar, conecte o Arduino novamente.
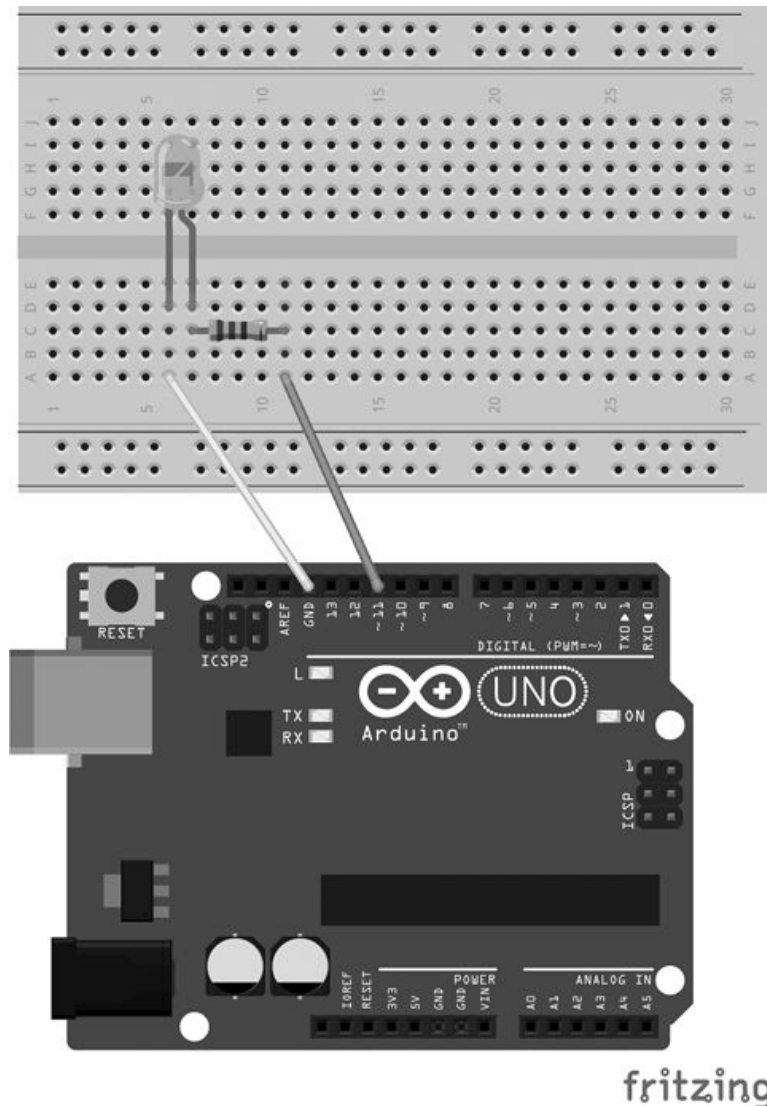
Figura 12.9 – Sketch do Fritzing para o projeto de LED piscante com o Arduino Uno.

A aplicação no Exemplo 12.1 expõe várias funções ao REPL por meio da função `REPL.inject()`. Isso significa que você pode controlar a placa quando estiver no ambiente REPL. Tudo que você tem a fazer é digitar o nome da função, como `on()` para acender a luz e `fadeOut()` para apagá-la, fazendo-a desvanecer.

Algumas das funções, como `pulse()`, `fadeIn()` e `fadeOut()`, exigem aquele pino PWM. A aplicação também utiliza uma animação para a função `pulse()`. Para interromper a animação, digite `stop()` no console REPL. Você verá a mensagem "Animation stopped". Ainda será necessário apagar o

LED depois de interromper a animação. E você deve parar as animações antes de apagar o LED; do contrário, elas não serão interrompidas.

Exemplo 12.1 – Aplicação interativa usando REPL para controlar o LED

```
var five = require("johnny-five");
var board = new five.Board();

board.on("ready", function() {
  console.log("Ready event. Repl instance auto-initialized! ");

  var led = new five.Led(11);

  this.repl.inject({
    // Permite um acesso de controle on/off limitado à
    // instância de Led a partir do REPL.
    on: function() {
      led.on();
    },
    off: function() {
      led.off();
    },
    strobe: function() {
      led.strobe(1000);
    },
    pulse: function() {
      led.pulse({
       easing: "linear",
       duration: 3000,
       cuePoints: [0, 0.2, 0.4, 0.6, 0.8, 1],
       keyFrames: [0, 10, 0, 50, 0, 255],
       onstop: function() {
       console.log("Animation stopped");
       }
      });
    },
    stop: function() {
      led.stop();
    },
    fade: function() {
      led.fadeIn();
    },
    fadeOut: function() {
      led.fadeOut();
    }
```

```
  });
});
```

Execute a aplicação usando Node:

```
node fancyblinking
```

Após ter recebido a mensagem "Ready event ...", você poderá executar comandos. Eis um exemplo, que cria o efeito de luz estroboscópica:

```
>> strobe()
```

Digitar `stop()` interrompe o efeito de luz estroboscópica, e `off()` apaga totalmente a luz. Experimente usar as funções `pulse()`, `fade()` e `fadeOut()`, mais sofisticadas, a seguir. A Figura 12.10 mostra o projeto enquanto a luz está pulsando (um pouco difícil de demonstrar em uma imagem estática – mas, acredite em mim, isso funciona).

Depois que tiver dominado as ações de fazer o LED piscar e pulsar, dê uma olhada em outras aplicações com Node/Arduino que você poderá testar:

- *Real-Time Temperature Logging with Arduino, Node, and Plotly* (Logging de temperatura em tempo real com Arduino, Node e Plotly, *http://bit.ly/1rZBQI8*)
- O *Arduino Experimenter's Guide to NodeJs* (Guia do NodeJS para quem quer experimentar o Arduino, *http://node-ardx.org/*) tem uma gama completa de projetos para testar, além de disponibilizar o código completo.
- *Controlling a MotorBot using Arduino and Node* (Controlando um MotorBot usando Arduino e Node, *http://bit.ly/1WH9Iqs*)
- Como alternativa ao Johnny-Five, experimente usar o módulo Cylon para Arduino (*https://cylonjs.com/*)
- *Arduino Node.js RC Car Driven with the HTML5 Gamepad API* (Carrinho de controle remoto controlado com Arduino Node.js usando a API HTML5 Gamepad, *http://bit.ly/1U4rwYE*)
- *How to Control Philips Hue Lights from an Arduino (and Add a Motion Sensor)* (Como controlar lâmpadas Philips Hue a partir de um Arduino [e adicionar um sensor de movimento], *http://bit.ly/1Veh5E5*).
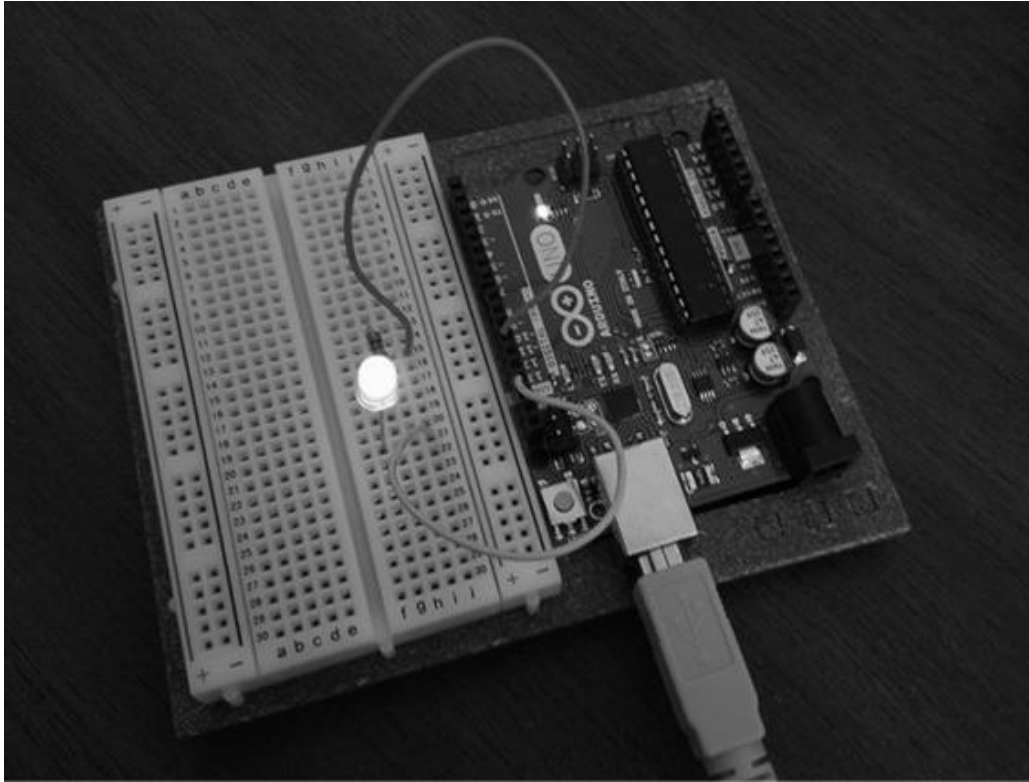
Figura 12.10 – Projeto de LED pulsante no Arduino Uno.

Você poderia se ocupar durante meses somente com o Arduino. No entanto, o Raspberry Pi acrescenta toda uma nova dimensão ao desenvolvimento de IoT, como veremos na próxima seção.

Node e Raspberry Pi 2

O Raspberry Pi 2 é uma placa muito mais sofisticada que o Arduino. Você pode conectar um monitor, um teclado e um mouse a ele e usá-lo como um verdadeiro computador. A Microsoft oferece uma instalação de Windows 10 para o dispositivo, porém a maioria das pessoas utiliza o Raspbian, que é uma implementação Linux baseada no Debian Jesse.

Conecte-se ao Raspberry Pi usando SSH

Você pode se conectar com o seu Raspberry Pi usando SSH se acrescentar um dongle WiFi. Eles são muito baratos e os que usei funcionaram de imediato. Depois que você tiver acesso à internet em seu Pi, o SSH deverá estar ativado por padrão. Basta obter o endereço IP do Pi e usá-lo em seu programa SSH. O WiFi está incluído por padrão no novo Raspberry Pi 3.

O Raspberry Pi 2 executa a partir de um cartão MicroSD. Ele deverá ter no mínimo 8 GB de tamanho e ser Classe 10. O site do Raspberry Pi

disponibiliza instruções de instalação (*https://www.raspberrypi.org/documentation/installation/*), mas recomendo formatar o cartão; copie o software NOOBs (New Out Of Box), que permite escolher o sistema operacional a ser instalado; em seguida, instale o Raspbian. Você também pode instalar diretamente uma imagem do Raspbian, baixado do site.

O Raspbian mais recente quando este livro foi escrito havia sido lançado em fevereiro de 2016. Ele vem com o Node instalado, pois também inclui o Node-RED: uma aplicação baseada em Node que permite literalmente arrastar e soltar o design de um circuito e ligar o Raspberry Pi diretamente a partir da ferramenta. No entanto, a versão do Node – v0.10.x – é mais antiga. Você usará o Johnny-Five a fim de utilizar o Node para controlar o dispositivo e ele deverá funcionar com essa versão, mas é provável que você vá querer fazer um upgrade no Node. Recomendo usar a versão LTS do Node (4.4.x quando este livro foi escrito) e seguir as instruções do Node-RED para upgrade do Node e do Node-RED manualmente (*http://nodered.org/docs/getting-started/installation.html*).

Confira o Node

O Node poderá estar instalado, mas com um nome diferente para o Node-RED. Se isso ocorrer, você deverá ser capaz de instalar uma versão mais recente do Node diretamente, sem precisar remover qualquer software.

Depois que fizer o upgrade do Node, instale o Johnny-Five. Você precisará instalar também outro módulo, o raspi-io: um plugin que permite que o Johnny-Five trabalhe com o Raspberry Pi.

```
npm install johnny-five raspi-io
```

Sinta-se à vontade para explorar o seu novo computador, incluindo as aplicações desktop. Depois que terminar, porém, o próximo passo será configurar o circuito físico. Para começar, desligue o Raspberry Pi.

Você precisará usar uma placa de ensaio para a aplicação de luz piscante "Hello World". Além do mais, será necessário ter um resistor, de preferência de 220 ohms, que é o tamanho de resistor geralmente incluído na maioria dos kits de Raspberry Pi. Deverá ser um resistor de quatro faixas: vermelho, vermelho, marrom e dourado.

Lendo as faixas dos resistores

A Digi-Key Electronics disponibiliza uma calculadora realmente prática e uma tabela de cores (*http://bit.ly/1YJxYpa*) para determinar o valor em ohms de seus resistores. Se tiver problemas para distinguir cores, você precisará da ajuda de um amigo ou de um membro da família, ou deverá usar um *multímetro* para medir o valor.

A Figura 12.6 mostra o sketch do Fritzing. Adicione o resistor e o LED à placa de ensaio, com o terminal cátodo (mais curto) do LED paralelo à última perna do resistor. Com cuidado, use dois dos fios que acompanham o seu kit de Raspberry Pi 2 e conecte um ao pino GRND (terceiro pino a partir da esquerda, na fila superior) e o outro ao pino 13 (sétimo pino a partir da esquerda, na segunda fila) na placa Raspberry Pi. Conecte as outras extremidades à placa de ensaio: o fio GRND será paralelo ao primeiro terminal do resistor e o fio do pino GPIO será inserido em paralelo ao terminal ânodo (mais longo) do LED.

Os pinos frágeis do Raspberry Pi

Os pinos são frágeis no Raspberry Pi, motivo pelo qual a maioria das pessoas usa um *breakout*. Um breakout é um cabo largo que se conecta aos pinos do Raspberry Pi e então é ligado à placa de ensaio. Assim, os componentes são conectados ao breakout em vez de se conectarem diretamente ao Raspberry Pi.

Ligue o Raspberry Pi novamente. Digite a aplicação Node. Ela é quase igual à aplicação para Arduino, mas você usará o plugin raspi-io. Além do mais, o modo de especificar o número dos pinos é diferente. No Arduino, você usou um número para o pino. No Raspberry Pi, você usará uma string. As diferenças estão em negrito no código:

```
var five = require("johnny-five");
var Raspi = require("raspi-io");
var board = new five.Board({
  io: new Raspi()
});

board.on("ready", function() {
  var led = new five.Led("P1-13");
  led.blink();
});
```

Execute o programa e o LED deverá piscar, exatamente como ocorreu no Arduino, conforme mostra a Figura 12.11.
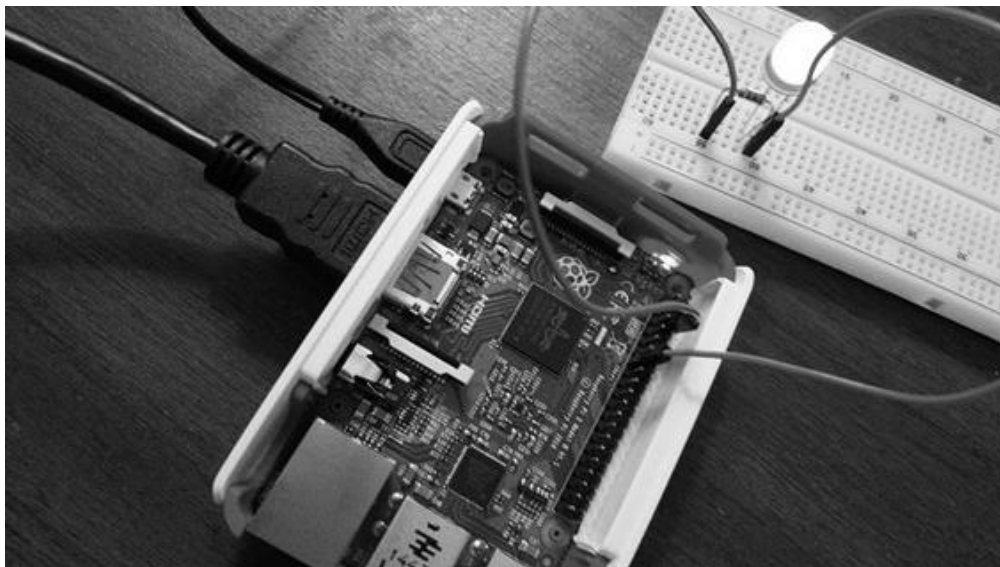
Figura 12.11 – Fazendo um LED piscar usando Raspberry Pi e Node.

Você também pode executar a aplicação interativa com o Raspberry Pi 2. Nessa placa, um pino PWM é a GPIO 18, que é o pino 12 para a aplicação Johnny-Five. É o sexto pino da linha superior à esquerda. Certifique-se de que o Raspberry Pi esteja desligado antes de mudar o fio do pino 13 para o pino 12. Não repetirei todo o código, mas a parte alterada da aplicação está incluída no bloco de código a seguir:

```
var five = require("johnny-five");
var Raspi = require("raspi-io");
var board = new five.Board({
  io: new Raspi()
});

board.on("ready", function() {
  var led = new five.Led("P1-12");

  // acrescenta animações e comandos
  this.repl.inject({
    ...
  });
});
```

Como o LED é maior, você poderá realmente ver melhor a animação quando digitar a função `pulse()`.

Outros projetos interessantes com Raspberry Pi e Node são:

- *Easy Node.js and WebSockets LED Controller for Raspberry Pi*

(Controlador de LED simples para Raspberry Pi usando Node.js e WebSockets, *http://www.instructables.com/id/Easy-NodeJS-WebSockets-LED-Controller-for-Raspberr/*)

- *Home Monitoring with Raspberry Pi and Node* (Monitoração de residência com Raspberry Pi e Node, *http://bit.ly/1YJyaVA*)
- *Heimcontrol.js: Home Automation with Raspberry Pi and Node* (Heimcontrol.js: automação de residência com Raspberry Pi e Node, *http://bit.ly/1rZC4PI*)
- *Build Your Own Smart TV Using RaspberryPi, NodeJS, and Socket.io* (Construa a sua própria smart TV usando RaspberryPi, NodeJS e Socket.io, *http://bit.ly/1WHapQw*)
- *Building a Garage Door Opener with Node and MQTT* (Construindo um abridor de porta de garagem com Node e MQTT, *http://bit.ly/240gbeR*) – usando um Intel Edison
- *Amazon's Guide to Make Your Own Raspberry Pi Powered Alexa Device* (Guia da Amazon para criar o seu próprio dispositivo Alexa com Raspberry Pi, *http://bit.ly/1TsoZHH*)

Há algo imensamente gratificante em ver uma reação física real e imediata às suas aplicações Node.

Sobre a autora

Shelley Powers tem trabalhado e escrito sobre tecnologias web – desde a primeira versão de JavaScript até o surgimento das ferramentas gráficas e de design mais recentes – há mais de doze anos. Seu livro mais recente da O'Reilly aborda web semântica, Ajax, JavaScript e web graphics. Shelley é uma ávida fotógrafa amadora e é aficionada por desenvolvimento web, além de gostar de aplicar seus experimentos mais recentes em muitos de seus sites.

Colofão

O animal na capa de *Aprendendo Node* é um rato-hamster (*Beamys*). Há

duas espécies de ratos-hamster: o rato-hamster maior (*Beamys major*) e o rato-hamster menor (*Beamys hindei*).

Os ratos-hamster habitam as florestas africanas do Quênia até a Tanzânia. Esse grande roedor prefere viver em ambientes úmidos: nas margens de rios e em áreas de florestas densas. Ele prospera em regiões costeiras ou montanhosas, embora o desmatamento venha ameaçando o seu habitat natural. Os ratos-hamster vivem em tocas com várias câmaras e são ótimos escaladores.

Esse roedor tem uma aparência muito distinta: pode ter cerca de 18 a 30 centímetros de comprimento e pesar até 150 gramas. Tem uma cabeça pequena e é coberto de pelos cinzentos, com uma barriga branca e uma cauda com manchas pretas e brancas. O rato-hamster, assim como outros roedores, tem uma dieta variável e apresenta bolsas nas bochechas para armazenar alimentos.

Muitos dos animais nas capas dos livros da O'Reilly estão ameaçados; todos eles são importantes para o mundo. Para saber mais sobre como você pode ajudar, acesse *animals.oreilly.com*.

UMA INTRODUÇÃO À PROGRAMAÇÃO DE COMPUTADORES
COM EXEMPLOS E EXERCÍCIOS PARA INICIANTES
Lógica de
Programação e
Algoritmos com
JavaScript
novatec
Edécio Fernando Iepsen

Lógica de Programação e Algoritmos com JavaScript

Iepsen, Edécio Fernando

9788575226575

320 páginas

[Compre agora e leia]

Os conteúdos abordados em Lógica de Programação e Algoritmos são fundamentais a todos aqueles que desejam ingressar no universo da Programação de Computadores. Esses conteúdos, no geral, impõem algumas dificuldades aos iniciantes. Neste livro, o autor utiliza sua experiência de mais de 15 anos em lecionar a disciplina de Algoritmos em cursos de graduação, para trabalhar o assunto passo a passo. Cada capítulo foi cuidadosamente planejado a fim de evitar a sobrecarga de informações ao leitor, com exemplos e exercícios de fixação para cada assunto. Os exemplos e exercícios são desenvolvidos em JavaScript, linguagem amplamente utilizada no desenvolvimento de páginas para a internet. Rodar os programas JavaScript não exige nenhum software adicional; é preciso apenas abrir a página em seu navegador favorito. Como o aprendizado de Algoritmos ocorre a partir do estudo das técnicas de programação e da prática de exercícios, este livro pretende ser uma importante fonte de conhecimentos para você ingressar nessa fascinante área da programação de computadores. Assuntos abordados no livro: Fundamentos de Lógica de

Programação Programas de entrada, processamento e saída Integração do código JavaScript com as páginas HTML Estruturas condicionais e de repetição Depuração de Programas JavaScript Manipulação de listas de dados (vetores) Operações sobre cadeias de caracteres (strings) e datas Eventos JavaScript e funções com passagem de parâmetros Persistência dos dados de um programa com localStorage Inserção de elementos HTML via JavaScript com referência ao DOM No capítulo final, um jogo em JavaScript e novos exemplos que exploram os recursos discutidos ao longo do livro são apresentados com comentários e dicas, a fim de incentivar o leitor a prosseguir nos estudos sobre programação.

específicos para administrar contêineres e interações com a plataforma. Padrões estruturais, que ajudam você a organizar contêineres em um Pod para tratar casos de uso específicos. Padrões de configuração, que oferecem insights sobre como tratar as configurações das aplicações no Kubernetes. Padrões avançados, que incluem assuntos mais complexos, como operadores e escalabilidade automática (autoscaling).

CANDLESTICK
Um método para ampliar lucros na Bolsa de Valores
novatec
Carlos Alberto Debastiani

Candlestick

Debastiani, Carlos Alberto

9788575225943

200 páginas

A análise dos gráficos de Candlestick é uma técnica amplamente utilizada pelos operadores de bolsas de valores no mundo inteiro. De origem japonesa, este refinado método avalia o comportamento do mercado, sendo muito eficaz na previsão de mudanças em tendências, o que permite desvendar fatores psicológicos por trás dos gráficos, incrementando a lucratividade dos investimentos. Candlestick – Um método para ampliar lucros na Bolsa de Valores é uma obra bem estruturada e totalmente ilustrada. A preocupação do autor em utilizar uma linguagem clara e acessível a torna leve e de fácil assimilação, mesmo para leigos. Cada padrão de análise abordado possui um modelo com sua figura clássica, facilitando a identificação. Depois das características, das peculiaridades e dos fatores psicológicos do padrão, é apresentado o gráfico de um caso real aplicado a uma ação negociada na Bovespa. Este livro possui, ainda, um índice resumido dos padrões para pesquisa rápida na utilização cotidiana.

AVALIANDO EMPRESAS
INVESTINDO EM AÇÕES
A APLICAÇÃO PRÁTICA DA ANÁLISE FUNDAMENTALISTA NA AVALIAÇÃO DE EMPRESAS
CARLOS ALBERTO DEBASTIANI
FELIPE AUGUSTO RUSSO
novatec

Compre agora e leia

Marcos Abe
MANUAL DE
ANÁLISE
TÉCNICA
ESSÊNCIA E ESTRATÉGIAS AVANÇADAS
TUDO O QUE UM INVESTIDOR PRECISA SABER PARA
PROSPERAR NA BOLSA DE VALORES ATÉ EM TEMPOS DE CRISE
novatec